2014

An Efficient Storage And Retrieval Mechanism For Large Unstructured Grids

Oyindamola Akande
University of Mississippi

AN EFFICIENT STORAGE AND RETRIEVAL MECHANISM FOR LARGE UNSTRUCTURED GRIDS

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the Computer and Information Science Department
The University of Mississippi

by

Oyindamola O. Akande

August 2014

ABSTRACT

The size of spatial scientific datasets is steadily increasing due to improvements in instruments and availability of computational resources. Scientific datasets today are often far too large to fit into a single machine's memory or even a single disk. However, much of the research on efficient storage and access to spatial datasets has focused on large multidimensional arrays. In contrast, *unstructured* grids consisting of collections of *simplices* (e.g. triangles or tetrahedra) present special challenges that have received less attention. Data values found at the vertices of the simplices may be dispersed throughout a datafile, producing especially poor disk locality. Partitioning multidimensional arrays across several machines or disks has become increasingly necessary. However, relatively little work has been done for unstructured grids.

We address this important problem of poor locality in two major ways. First, we reorganize the unstructured grid to improve locality in both the dataset space and in the data file on disk using a specialized chunking approach that maintains the spatial neighborhood relationships inherent in the unstructured grid. This reorganization produces significant gains in performance by reducing the number of accesses made to the data file. We examine the effects of different chunking configurations on data retrieval performance. A major motivation for reorganizing the unstructured grid is to allow the application of *iteration aware prefetching*.

Second, we describe a prefetching method that takes advantage of prior knowledge of the user's access pattern. Applying this prefetching method to unstructured grids produces further performance gains over and above the gains seen from reorganization alone.

In addressing the poor locality, we investigated partitioning unstructured grids at the disk level and its effect on overall system performance. We build upon this and investigate the

effect of an *in-memory partitioning* performed on top of the existing *disk level partitioning*. We also examine the performance benefits of declustering unstructured grids across several disks. Given this declustered dataset, we describe and explore a parallel data retrieval method that takes advantage of prior knowledge of a user access pattern. Our test results demonstrate very significant performance gains. Lastly, we present guidelines for choosing effective partitionings of datasets when the access pattern is known in advance.

DEDICATION

To my parents, my siblings and to my fiancé.

ACKNOWLEDGEMENTS

I thank God for granting me the grace and opportunity to complete this work.

I would like to express my sincere appreciation to my advisor, Dr Philip J. Rhodes. I am deeply indebted to him for his kindness and support throughout my Ph.D. program. I also thank my dissertation committee members Dr. Gregory L. Easson, Dr. Feng Wang and Dr. Byunghyun Jang for their time and invaluable comments during my dissertation review.

I would like to express many thanks to the chair of the Computer Science department, Dr. H. Conrad Cunningham, and indeed all members of the Computer Science faculty and staff for their selflessness and encouragement.

My most profound gratitude goes to my parents, Professor and Mrs S. O. Akande for their love, support and motivation. You have always stood as a pillar of support and encouragement, to which I am eternally grateful. I would also like to thank my siblings for their ever present love and understanding.

Finally, this work would not be complete, but for the undying love and support of my fiancé throughout my graduate program. I am grateful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

MOTIVATION

With increased processing power and vast storage space available at an affordable price, the size of scientific data generated from scientific simulations and real world applications continue to grow at an exponential rate. A typical dataset size is in the Terabyte and Petabyte scale. Developing efficient means of storing and retrieving the large amount of data that is generated, without massive duplication of data is however not growing at an equivalent rate. Efficiently storing and retrieving large data volumes is an important goal for the scientific data community. *Spatial* datasets present special challenges because elements nearby in the data space may be far apart in the file on disk. The overall size of the data thus necessitates dividing the data across multiple disks on a single machine or across multiple machines.

Sample points in *unstructured* grids are placed arbitrarily throughout the dataset domain and have some arbitrary number of neighbors. Because there is no pattern to either the *geometry* or *topology* of an unstructured grid, they must be represented explicitly. Rather than listing neighbors directly, unstructured grids are commonly organized into non-overlapping *cells*. A scientific application will often need to query the dataset for an arbitrary location within the domain, rather than restricting queries only to locations corresponding to sample points. Cells are essential for this purpose, because the cell containing the query

location is used to *interpolate* a value from the data associated with the cell vertices. Unfortunately, this operation can be expensive for two reasons. First, finding the cell that contains the query location can be computationally expensive because a large number of cells must be tested before the containing cell is found. Second, there is significant I/O cost incurred when reading the data values associated with the cell vertices. Data values may be spread over several distant locations in the data file, exhibiting poor locality and requiring separate read transactions to the underlying storage device.

In order to efficiently solve the problem of locating a cell due to the extremely large dataset, we require a mechanism that will speed up the process of identifying the candidate cell and reading the cell from disk into memory. The approach used in this work is to split up the large dataset into chunks. Partitioning datasets into chunks can cause difficulties when cells span partition boundaries. It is desirable to make each partition self-sufficient, but duplicating dataset values between partitions is an unattractive option for large datasets. An important goal of this research is to prevent excessive duplication of dataset values across partition boundaries and provide a way to choose the coarseness of the resulting grid after partitioning.

Due to the nature of regular datasets, partitioning the data is relatively straight forward and subsets of a much larger dataset can be easily read, unlike for unstructured grids that have dataset points dispersedly distributed within the data. Prior work [69, 68] has investigated selecting and reading partial replicas of a large regular dataset. We want to apply such work to unstructured grids that have data distributed across multiple machines spanning different geographic locations. This work investigates partitioning of the data from a single machine perspective and is being developed as part of a much larger system such as a distributed environment in which the partitions reside on multiple machines spanning different geographic locations.

The traditional approach is to store such datasets at single sites, and to perform computation on hardware immediately next to the data. The size of a dataset or computation

is then limited by the resources available at a single site, even if suitable resources are available elsewhere, and even if only a small portion of the data is actually used. We would like to explore ways to escape such limitations by allowing computation to efficiently access spatial data that is not necessarily local, and by breaking very large datasets into multiple *partial replicas* that can be stored at various sites [69, 68].

We perform a further partitioning in memory of the resulting grid. Determining what effects the granularity of the in memory partitioning has on the overall system performance and the manner the data will be accessed is a major motivation of our research. We focus our efforts, influenced by our results, and prior knowledge of how the data will be accessed to determine the best way to partition the dataset to facilitate efficient access.

In dealing with unstructured grids, a tempting approach is to simply resample the data points of the unstructured grid to make it regular, and apply methods used for regular datasets. This approach will create more problems than solution for large unstructured grids. Most scientific datasets have data values associated with the sample points and resampling the data will create extra vertex points and associated data values, which in turn will increase the overall size of the dataset. Regular grids do not preserve the adaptive resolution that unstructured grids naturally provide. We do not resample the unstructured grid data because for these reasons and instead strive to improve the storage representation of the unstructured dataset.

One of the main motivation is to present a method for dramatically enhancing I/O performance with unstructured grids by improving locality while correctly handling the problem of triangles or tetrahedra that span chunk boundaries (inherent in the unstructured grid is the tightly knit relationship that exists between sample points of the domain). We address and improve the storage efficiency of unstructured spatial data and improve the access time.

A major goal of the work described in this paper is the extension of Iteration Aware Prefetching (IAP) to the much more difficult case of unstructured grids and fitting unstructured grid to the rod storage model, which allows a series of adjacent elements to be read

from file in a single read transaction. Past work [56] has applied IAP caching mechanism to files that fit the rod storage model, including both linear and chunked [63] files. In doing so, we can achieve very significant gains in performance by taking advantage of prior knowledge of the access pattern. Merging the data into a single file helps us take advantage of *filesystem prefetching*[65], which is only effective when reading from a single file. Merging the cell groups into a single file also improves the locality of reference in the one dimensional file space on disk. Many computations access data in the same manner regardless of the data *values* themselves. For example, the manner in which visualization applications access data often depends only on view direction and similar factors, rather than the values being visualized. Our previous efforts were implemented for regular datasets and we would like to extend this work to unstructured grids.

Declustering the data and sharing it among several disks and analyzing performance is an approach we investigate. When the partitioning is very fine grained, our goal is to improve the overall performance time by amortizing latency costs over multiple partitioning elements we access in parallel on multiple disks, instead of paying similar costs for each individual partitioning element. Taking advantage of the parallelism provided by multiple disks is an attractive option and is a main focus in our work. We hope to extend this method to a distributed environment, so that even the size of local storage no longer limits feasible dataset size.

We target some of the most important features of the *Big Data* problem. Our splitting method addresses unstructured spatial sets with very large *volume*, allowing access to data sets much larger than memory. Our prefetching technique addresses *velocity*, greatly improving the speed of data access by reducing the frequency of read operations. Lastly, the work described here is being conducted as part of a larger system which handles many different *varieties* of spatial scientific data, while providing scientists with a single intuitive interface.

CHAPTER 2

OVERVIEW

Due to the overwhelming size of modern datasets, and that complete datasets will no longer fit in memory, modern systems have resolved to dividing the data and storing it locally or across multiple machines that may span multiple geographic locations. Typical scientific data sets has grown to the terabyte and even petabyte scale with increased processing power and vast storage space available at an affordable price. Efficiently storing and retrieving large data volumes is an important goal for the scientific data community. *Spatial* datasets present special challenges because elements nearby in the spatial domain are not necessarily close in the file representation on disk.

This problem is further compounded with *unstructured* grids or meshes [43, 40]. The term "unstructured" is overloaded, but in this work, it refers to data in which the spatial relationship between dataset elements does not follow a regular pattern. We use the term unstructured grid and unstructured meshes interchangeably. Unlike array based data, unstructured grids require that the location of sample points in the dataset's spatial domain (i.e. the *geometry*) be explicitly represented as a list of coordinate tuples. Similarly, the neighborhood relationship between sample points (i.e. the *topology*) must be represented as a list of simplicial *cells*, typically triangles or tetrahedra.

In figures 2.1 and 2.2, we show two unstructured spatial datasets. The image shown in

figure 2.1 shows an unstructured grid representation of an F-16 aircraft containing about 1.4 million cells and over 250,000 vertices. Figure 2.2 is generated from simulation of the 1994 Northridge earthquake in the greater LA basin [41]. This data contains 11.5 million hexahedral elements. These are a few application areas of large unstructured grids in scientific research.

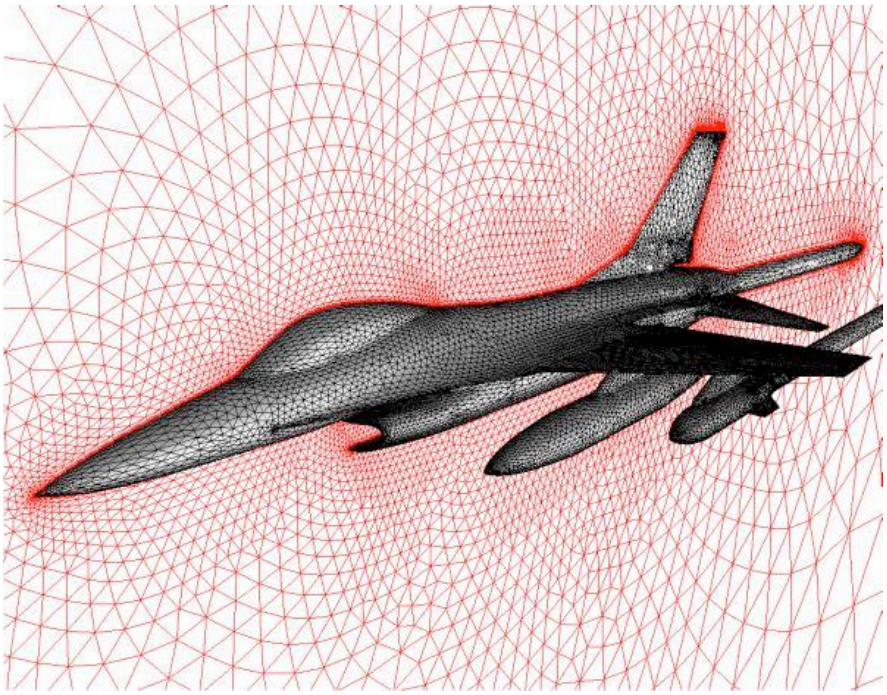

Figure 2.1. Unstructured tetrahedral grid of F-16 aircraft produced at the NASA Langley Research Center [5]

Cells are essential for interpolating data values for locations that don't correspond to sample points, so efficiently retrieving the data belonging to a cell is of primary importance. Unfortunately, the data values corresponding to cell vertices may be spread throughout the data file, perhaps resulting in several file accesses with poor locality.

We develop a mechanism that provide efficient means of storing and retrieving data while minimizing duplication across partition boundaries. In order to achieve data availability, the data is usually replicated across multiple nodes. The data satisfying a user query, and that can be quickly located is thus used to resolve such queries. Duplication increases the overall size of the dataset. Duplication in order to efficiently resolve query is good but,

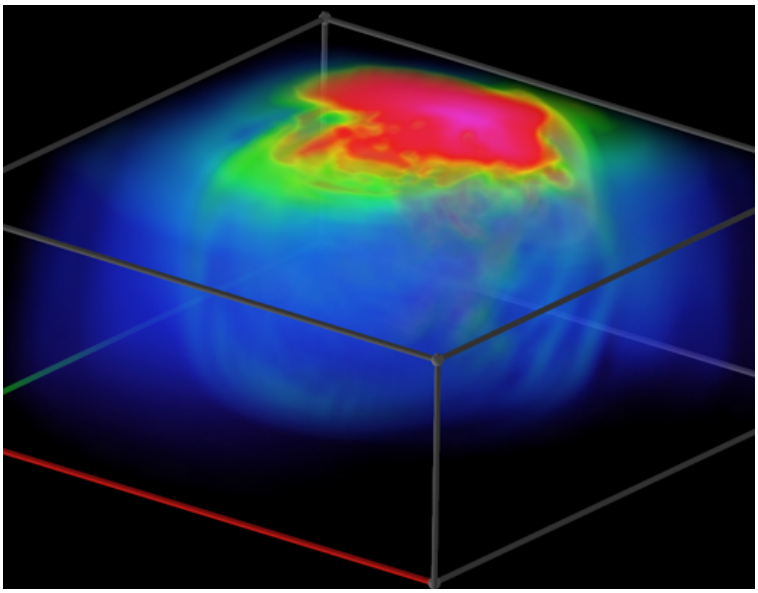

Figure 2.2. Visualization of earthquake simulation courtesy Visualization and Interface Design Innovation (ViDi) research group [7]

less duplication across partition boundaries to resolve the same query is even better. With structured multidimensional data, partitioning the data is less complicated as point location can be easily calculated using array indices. unstructured grids presents additional challenges that makes the partitioning more complicated.

This dissertation presents a method for dramatically enhancing I/O performance with unstructured grids by splitting the data into chunks with improved locality while correctly handling the problem of triangles or tetrahedra that span chunk boundaries. While considerable work has been done in the area of graph partitioning [31], we concentrate on scientific data described by a collection of simplices. Our solution does not require potentially expensive duplication of data between chunks. Extending previous work [56] to apply to unstructured grids, we further improve performance using a prefetching cache that takes advantage of prior knowledge of an application's access pattern. Many computations access data in the same manner regardless of the data *values* themselves. For example, the manner in which visualization applications access data often depends only on view direction and similar factors, rather than the values being visualized.

We investigate and develop an efficient means of partitioning large unstructured grids at different levels of the memory hierarchy that is, on disk and in memory. We provide efficient means of data retrieval when accessing data using a Least Recently Used (LRU) cache using the separate *reorganized files* generated from the partitioned dataset, a merged file that improves data locality in the underlying one dimensional file and *iteration aware prefetching (IAP)* on the merged file. We also investigate the effects and improvements of applying declustering mechanism to unstructured grids and use IAP to access data in parallel.

The need for faster access to stored data has led to declustering schemes that not only provide parallel access to data, but also replicate data across multiple disks for fault tolerance.

CHAPTER 3

GRANITE SYSTEM

Scientific data sets are represented in a variety of formats, stored on local disk or distributed across a collection of machines. There are many examples of middleware that abstract away the details of storage, allowing users to focus on the science [3, 2, 4]. The *Distributed Spatial Computation and Visualization Environment (DISCoVer)* focuses on providing a convenient model for *spatial* scientific datasets, allowing scientists to query datasets naturally using the domain space, shielded from the minutiae of representation. DISCoVer consists of several components that support both distributed and local access to large spatial datasets.

The *Granite* component of DISCoVer provides efficient access to spatial datasets stored on local or remote disks [56, 55]. While much of our previous work on Granite has focused on *regular* datasets, the work described here is applied to *unstructured* grids, which presents special challenges.

3.1 Regular Data

The placement of sample points within the domain of a regular data set follows a repeating pattern that typically allows point locations to be calculated from array indices rather than stored explicitly. Similarly, each point's relationship with its neighbors is uniform across the dataset, except perhaps at the boundaries. We refer to the placement of sample points

within the domain as the *geometry*, and the neighborhood relationship that exists between sample points as the *topology*.

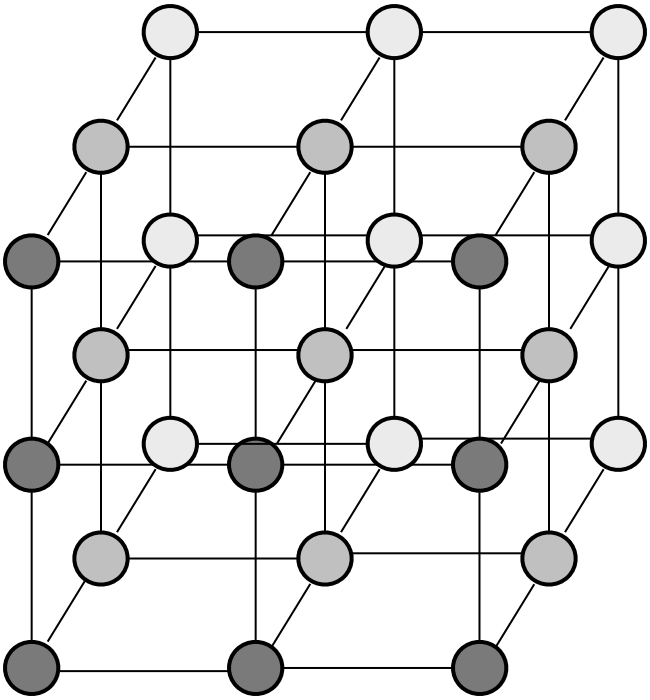

Figure 3.1. Regular Data

A common example of a regular dataset is a three dimensional grid, where sample points are evenly spaced, and have neighbors to the north, east, south, west, front, and back as shown in figure 3.1.

3.2 Unstructured Grids

In contrast to regular data, sample points in unstructured grids are placed arbitrarily throughout the dataset domain and have some arbitrary number of neighbors. Figure 3.2 depict unstructured grids. Because there is no pattern to either the geometry or topology of an unstructured grid, they must be represented explicitly. That is, the geometric coordinates of each sample point must be stored directly in the file. Similarly, representing an unstructured topology requires listing the neighbors of each sample point.

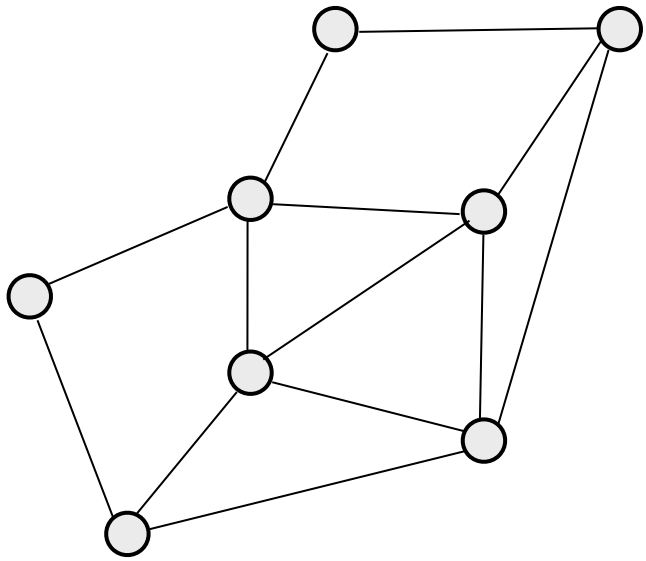

Figure 3.2. Unstructured Grid

Rather than listing neighbors directly, unstructured grids are commonly organized into non-overlapping *cells*, where each cell is a convex region bounded by a set of sample *points*, also referred to as *vertices*. Cell shapes are often *simplices*, meaning that the number of vertices is minimal for a given dataset dimensionality. For two dimensional datasets, simplicial cells are triangular, while for three dimensions, a tetrahedron is used. In any case, the dataset can be organized using three lists. A vertex list contains the sample point locations, while a cell list denotes each cell using a series of indices into the vertex list. Data values associated with vertices can be stored in a separate data list, or included directly in the vertex list. Figure 3.3 shows an example for a 2D dataset consisting of triangular cells.

A scientific application will often need to query the dataset for an arbitrary location within the domain. In this work, we refer to such query as a *datum query*. Rather than restricting queries only to locations corresponding to sample points. Cells are essential for this purpose, because the cell containing the query location is used to *interpolate* a value from the data associated with the cell vertices. Unfortunately, this operation can be expensive for

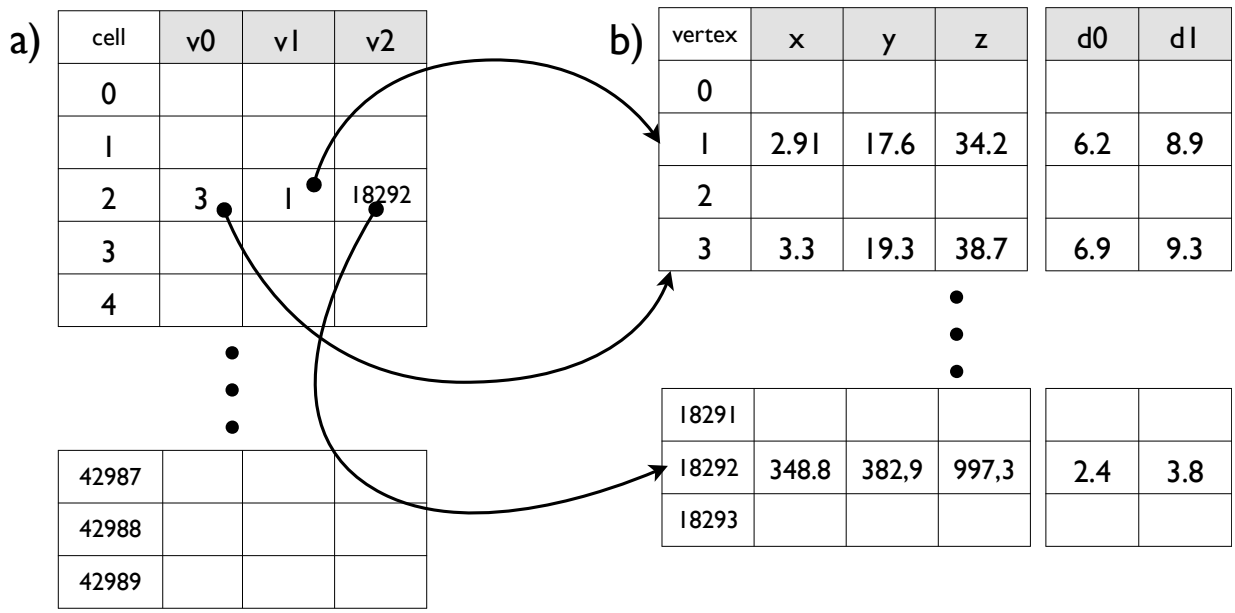

Figure 3.3. Unstructured grids are organized into lists of cells (a) consisting of references into a list of vertices and associated data (b).

two reasons. First, finding the cell that contains the query location can be computationally expensive because a large number of cells must be tested before the containing cell is found. Second, there is significant I/O cost incurred when reading the data values associated with the cell vertices. As shown in figure 3.3, data values may be spread over several distant locations in the data file, exhibiting poor locality and requiring separate read transactions to the underlying storage device.

3.3 Granite

Granite is a middleware package that provides convenient access to large spatial scientific datasets. The Granite system is composed of two main layers: the *datasource* and the *lattice* layer. The datasource layer supports array based regular data, while the lattice layer adds support for unstructured grids.

Prior work [56, 55] describes Granite's unique *Iteration Aware Prefetching (IAP)* mechanism, that creates n-dimensional cache blocks with a shape that is tuned to the user

access pattern. The user represents the access pattern in advance as an iterator, after which the Granite system reads the underlying dataset via the cache blocks, which vastly reduce the number of reads made to disk.

Those previous efforts were implemented for regular datasets using Granite's data-source layer. A major goal of the work described in this work is the extension of IAP to the much more difficult case of unstructured grids. In order to facilitate our explanation of this work, we now describe some granite specific terminologies and how it relates to well established spatial data concepts.

3.3.1 Lattice

The lattice layer provides a logical representation of an unstructured grid. The lattice layer represents topology and geometry information and also provides both topological and geometrical views of the dataset.

A lattice is able to associate a data value with any location in the dataset domain, regardless of whether a sample point actually exists for that location. In the very common case where a query point does not correspond to a sample point, the data value (i.e. datum) must be generated using an *approximator* function that computes the value from the data values associated with the vertices of the cell containing the query point.

If cells were simply contained in a single list for the entire dataset, a very large number of containment tests would have to be performed in order to find the cell that contains the query point. Instead, we reorganize the dataset into a grid of *cell groups*, where each cell group corresponds to a *partition element*, and contains all cells that intersect with that element. When processing a datum query, the lattice can easily map the query point to the partition element that contains it, and load only that element from disk. It therefore has a much smaller set of candidate cells to examine for containment, and each cell group can be read from disk into memory with a single read transaction, vastly reducing the number of I/O operations required for a lattice datum query.

Since the grid of cell groups serves as a spatial data structure [27] that speeds the search for the cell containing the query point, we can use yet another grid of cell groups in memory to further speed the search once a cell group has been loaded from disk. That is, the cell group loaded from disk actually contains an in memory grid, which once again partitions the cells into a grid of cell groups that further reduce the number of containment tests required.

Once the cell containing the query point has been found, it remains only to apply the approximator function to produce a data value. The function takes the data values associated with the cell vertices and the position of the query point within the cell, and computes a data value to be returned as a result of the query. Many options exist for this function, and the scientific user is allowed to specify a function suitable for their particular purposes.

3.3.2 Index Space

An *index space* is an abstraction of the discrete space formed by array indices. An index space may be of any dimensionality, and is addressed by cartesian coordinates with integer values. Granite's *IndexSpaceID* class encapsulates these coordinates.

Hyper-rectangular sub-regions of an index space are represented using the *ISBounds*, which uses two IndexSpaceID objects to represent two corners of the region. Such regions can only be *isoaligned*, meaning that the sides are always parallel to the major axes of the index space, similar to the *Axis Aligned Bounding Box (AABB)* commonly used in computer graphics and related fields [73].

3.3.3 Partitionings

A *partitioning* or *grid* divides a space into a set of *partition elements* which can be viewed as an index space. For example, a two dimensional *RegularISPartitioning* divides an index space into equally sized rectangles, which in turn form another index space. Figure 3.4

shows a domain split using a regular partitioning. This kind of partitioning plays a key role in the dataset splitting method described in section 4.1. Non-regular partitionings are also certainly possible, and may be useful for load balancing and variable resolution applications in future work.

The shape of the partitioning determines the number of partition elements created. Figure 3.4*a* shows a partitioning using a 6×6 ISBounds that creates 36 partition elements and figure 3.4*b* uses a 21×4 ISBounds that creates 84 partition elements.

3.3.4 Geometric Space

For datasets with sample points located in a continuous geometry, we provide facilities similar to those offered for the discrete index space. The *Point* class denotes a location in a continuous space, while the *GBounds* denotes an isoaligned hyper-rectangular region. *RegularGPartitioning* partitions a continuous space in a manner analogous to the RegularISPartitioning class described above.

Creating an efficient partitioning for the dataset is important as it will determine the number of *cell groups* created and the eventual cache shape and size. We discuss these issues later in the paper.

3.3.5 Iterators

Because a major goal of DISCoVer is to improve I/O performance using knowledge of the access pattern, we use iterators both to represent the access pattern and to perform the actual iteration through the datasource index space. Iterators have a value that changes with each invocation of the iterator's *next()* method, and this value can be used directly in both datum and subblock queries. The *iteration space* is the space traversed by the iterator. It may be the entire index space of a datasource, continuous domain of a lattice, or some subset of that space. The iterator's constructor accepts a reference to the iteration space and specifies the region to be traversed. This is particularly useful for region queries

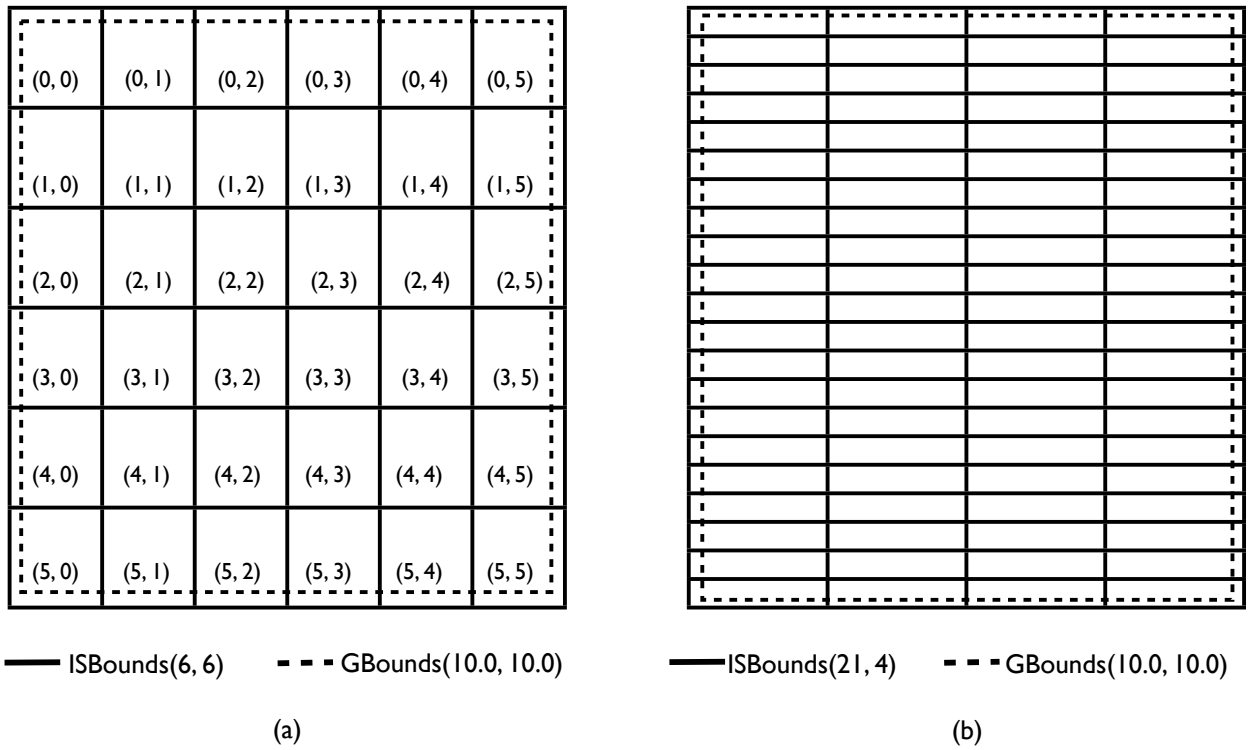

Figure 3.4. Dataset with an upper GBounds of (10.0, 10.0) showing partitioning using (a) an ISBounds of (6,6) and (b) an ISBounds of (21,4)

and supports access patterns such as *arbitrary rectangular subset*, an *arbitrary area on an orthogonal plane*, a sub area or the whole domain [39].

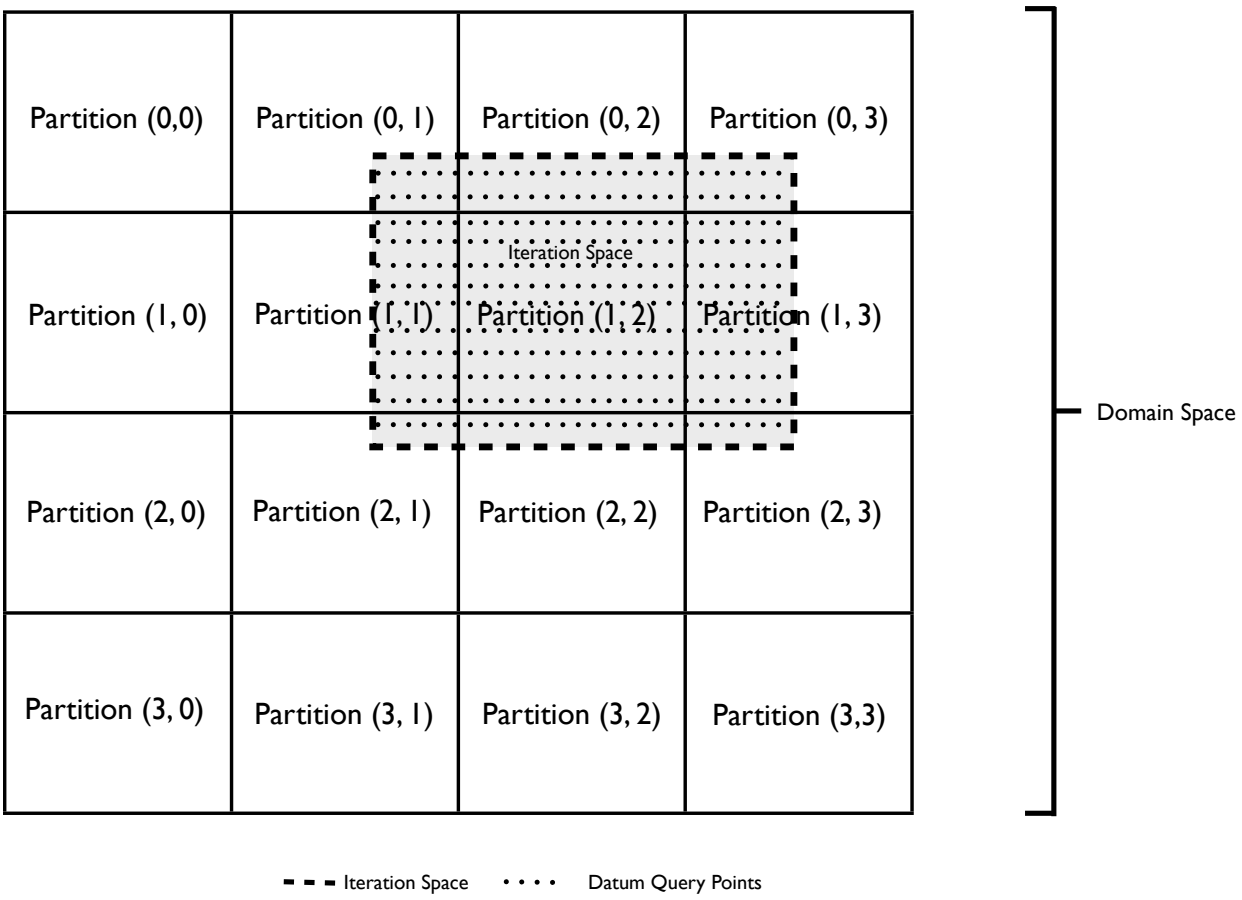

Figure 3.5. Iteration space spanning a subset of the geometric domain space showing query points. The domain has been partitioned using a 4×4 ISBounds.

Figure 3.5 shows a data space that has been partitioned using a 4×4 ISBounds. The shaded region is an iteration space whose limits are specified by a GBounds given to the iterator constructor. In this case, the iterator will iterate over a subset of the domain space. The iterator performs a datum query at evenly spaced locations as it iterates through the space. In the case shown, the spacing between locations is quite fine compared to the partitioning, so the iterator visits every partition within the iteration space. With a coarser iterator spacing, some partitions might be skipped entirely by the iterator.

3.3.6 Axis Orderings

An *axis ordering* is an ordered list of axes, where each axis is identified by an integer. Many iterators use axis orderings to denote their most (and least) rapidly changing axes, specifying the manner in which they proceed through space. For example, *iteration ordering* {2,1,0} describes an iteration that changes most rapidly along axis 0, and least rapidly on axis 2. In this example, axis 0 can be called the *run axis*, while axis 2 could be called the *slice axis*. This is particularly useful in visualization applications where data access is dependent on the view direction. The diagram in figure 3.6 shows the run axis and the slice axis for a 3D dataset. The image in figure 3.7 uses axis ordering {0, 1, 2} and figure 3.8 uses axis ordering {1, 2, 0} which varies the view direction on the same dataset.

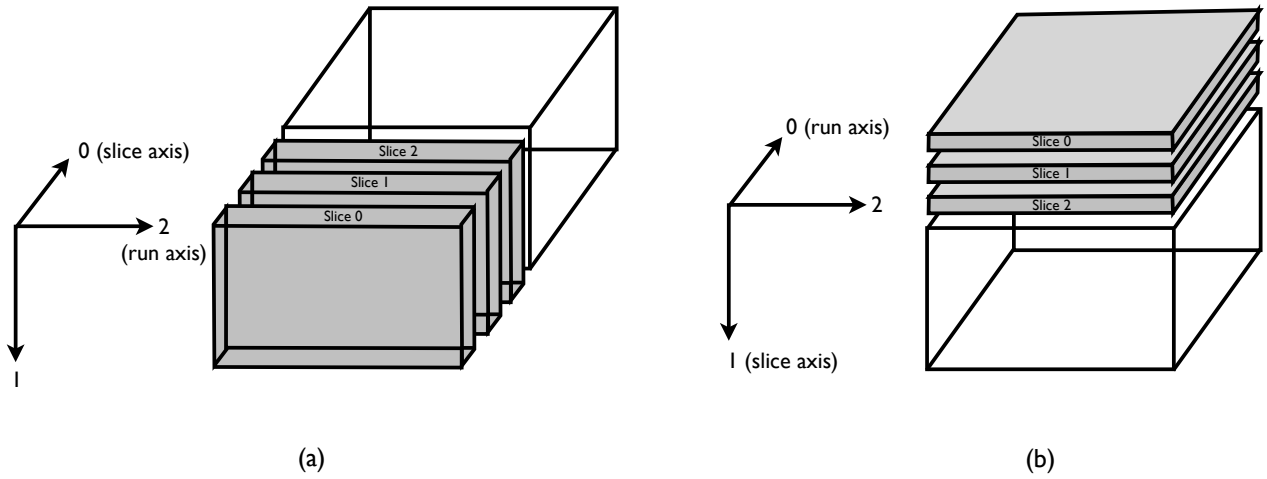

Figure 3.6. (a) Axis ordering {0, 1, 2} (b) Axis ordering {1, 2, 0}

For some types of file, axis orderings are also used by the datasource layer to map an index space to the one-dimensional file space. This is known as a *storage ordering*.

3.4 Iteration and Storage Orderings

Consider the diagram shown in figure 3.9*a* where the user's ordering and storage ordering are similar. In this case the iterator accesses data in the order it was stored. The access

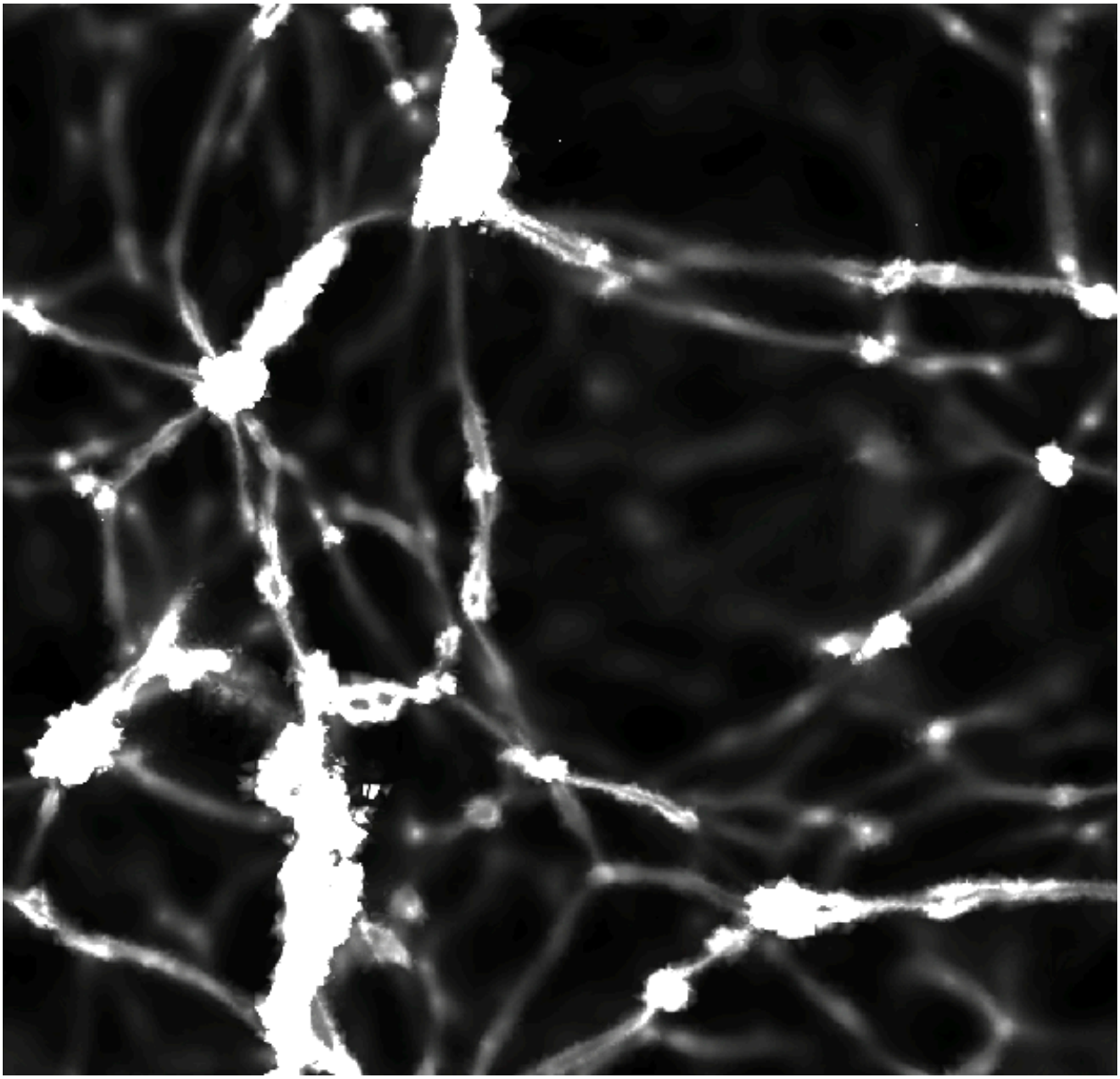

Figure 3.7. Internal energy in the Arepo data at z=0.5 using axis ordering (0, 1, 2)

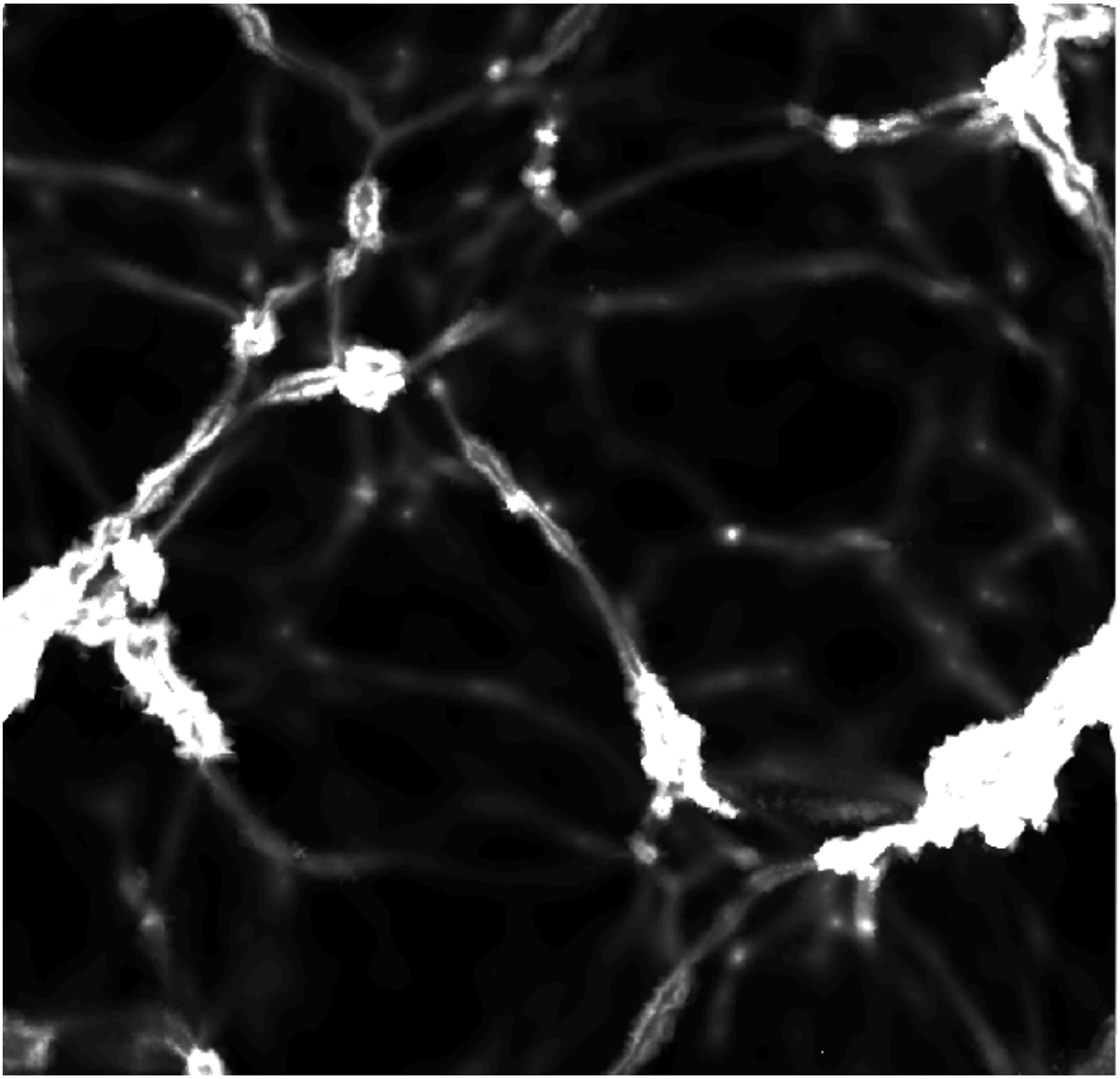

Figure 3.8. Internal energy in the Arepo data at z=0.1 using axis ordering (1, 2, 0)

pattern is contiguous both spatially and in the one dimensional file, so the filesystem cache is very effective.

When the user's ordering is different from the storage ordering as shown in figure 3.9*b*, performance will be poor for an iteration that accesses the data in a column order fashion. We remedy this problem using the prefetching method described in section 6.3.

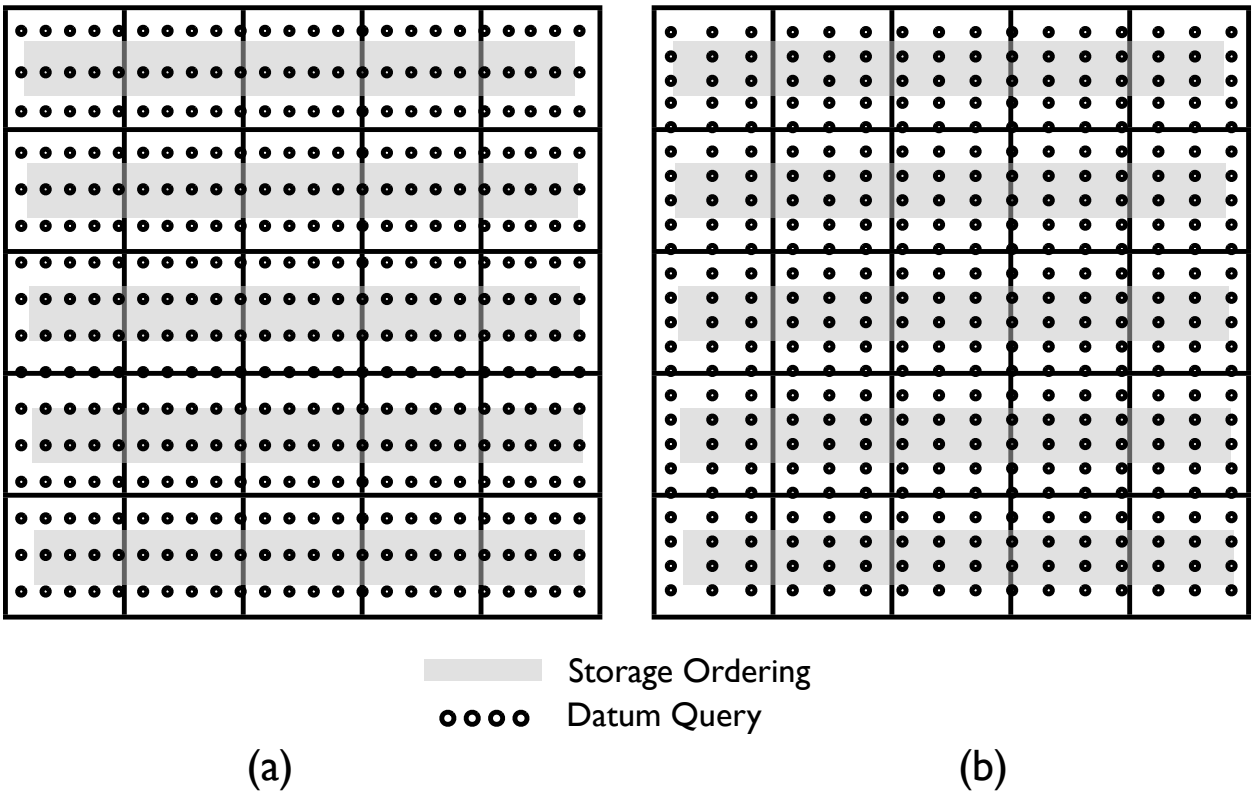

Figure 3.9. (a) Access ordering {0,1} of datum query and storage ordering {0, 1} (b) Access ordering {1, 0} and storage ordering {0, 1}

Figure 3.6 shows three dimensional access orderings. The access pattern in this case are synonymous with planes. In figure 3.6*a*, the ordering {0, 1, 2} means the most rapidly changing axes are 2, 1 and 0 respectively.

3.5 Partition Shapes

Another important factor for performance is the shape of the partition elements. Figure 3.4 shows two possibilities for a two dimensional dataset, while figure 3.10 shows some options for the three dimensional case. Figure 3.10*a* shows a cubic partition element and the other forms are as shown in figures 3.10*b*, 3.10*c* and 3.10*d*.

In our results, we present how performance for a given iteration order affects the storage ordering and partition element shape.

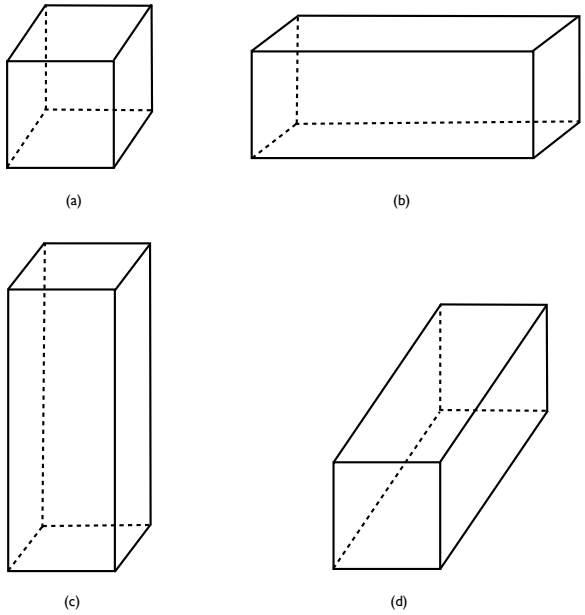

Figure 3.10. 3D Partition Elements Shapes

CHAPTER 4

DATA REORGANIZATION

Chunking [63] is the process of dividing a much larger volume into smaller multidimensional tiles that are stored and accessed together. It is a very effective general-purpose technique for improving access to multidimensional arrays stored as files. Data that is nearby in the data space is stored together on disk, improving performance through improved locality of reference. Chunking has been applied to regular datasets very successfully for decades [19, 60], but it is not as easily applied to an unstructured grid.

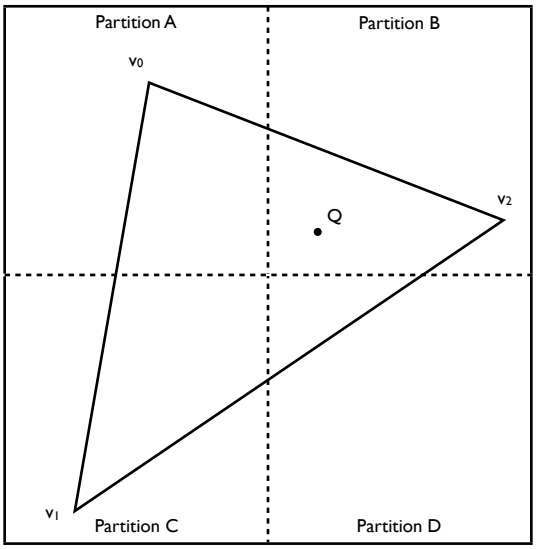

Figure 4.1. A cell spanning multiple partition elements

The main problem is cells that span boundaries. Such cells must be members of more than one chunk (or partition element), which could require expensive duplication of data. Consider the diagram shown in figure 4.1 in which a cell made up of vertices v_0, v_1 and v_2 span multiple partitions. Several data values such as mass, density and pressure, may be associated with each vertex. This data likely represents the majority of the storage consumed by the file. During data access, assuming only partition B is loaded into memory and location Q is queried for density stored at that location. Resolving the query involves interpolating the density values stored at v_0, v_1 and v_2. The neighborhood relationship between vertex v_2 and adjacent vertices v_0 and v_1 would be lost except that we develop a means of maintaining the topology information across partition boundaries. A naive approach would duplicate vertices v_0 and v_1 and the data associated with them and store it all in partition B. The approach adopted by the Granite system requires no such duplication.

The remainder of this section describes the splitting process and the storage model we applied to improve locality within the one dimensional file.

4.1 Splitting The Unstructured Grid

In order to split the unstructured grid, we first construct a regular partitioning of the geometric domain, forming an index space for which each location corresponds to an isoaligned n-rectangular region called a *partition element*. The splitting process consists of reading each cell from the original data file and determining which partition element contains it. The simplest case occurs when a cell is wholly contained by a partition element or wholly outside the partition element. Cells that span partition boundaries present additional challenges. We use the test provided in Akenine-Möller's paper [10] to detect the boundary cases.

Figure 4.2*a* shows a lattice representation for a 2D dataset before it was split and figure 4.2*b* shows the same dataset after it has been split into separate partition elements. We refer to the file representation of the partition elements on disk as the *reorganized files*.

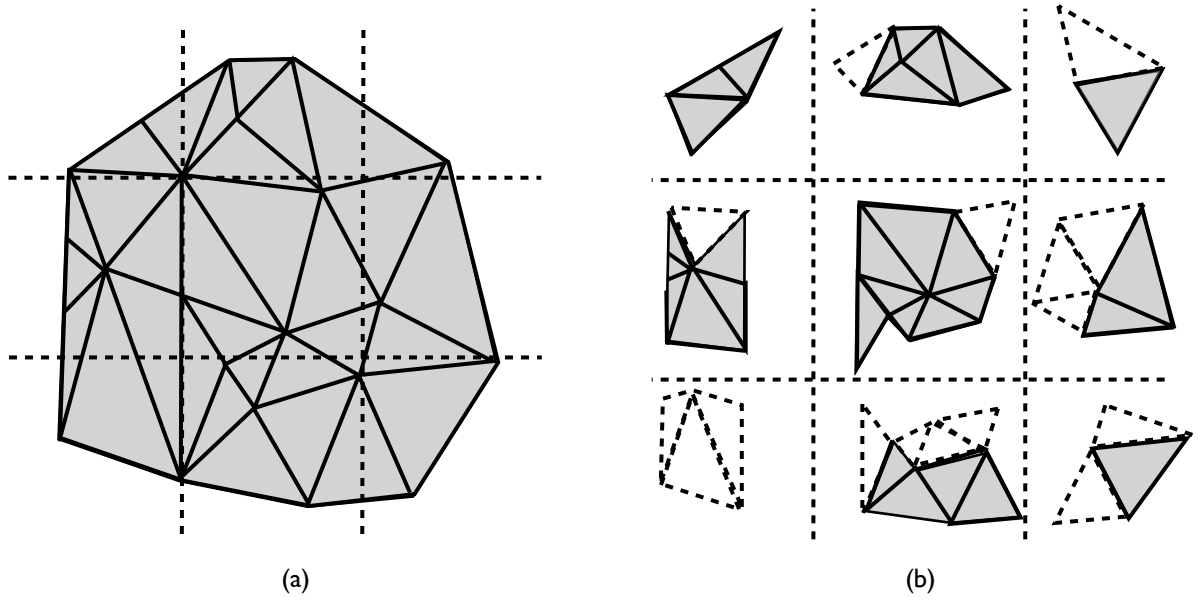

Figure 4.2. Splitting an unstructured 2D lattice. Many cells span partition boundaries, and are shown several times in (b). Partitions containing shaded cells in (b) are responsible for representing all data for those cells, but unshaded cells represent only references to cell data.

4.2 Cell Groups

Each partition element corresponds to a *cell group* which maintains the set of cells that intersect that particular partition element. Each cell group contains the vertex information and data for cells that are contained within that cell group. The data for cells that span the partition boundary are treated differently.

Due to the size of the datasets we are handling, cell groups need to be occasionally committed to disk while splitting is being performed. Each cell group is written to its own separate file, which allows a cell group to be flushed easily to disk as memory limitations demand.

4.3 Owned and Borrowed Cells

Partitioning datasets into chunks can cause difficulties when cells span partition boundaries. It is desirable to make each partition self-sufficient, but duplicating data between partitions

is an unattractive option for large datasets. Our solution to this problem is a compromise in which the vertex coordinates for all intersecting cells are stored in each partition's cell list, but the data associated with the cell vertices is stored in only one partition. This partition is called the *owner* of the cell. All cells have exactly one owner. Partitions that intersect with a cell that is owned by another partition are referred to as *borrowers* of the cell, as shown by the shaded cells in figure 4.2*b*. A cell may have any number of borrowers, always one less than the number of partitions it intersects with. A cell that has no borrowers is entirely contained within its owner partition. Borrowed cells are represented on disk using only the vertex indices necessary for representing the cell, and an identifier that denotes the owner partition and position in the owner cell list. Since all cells have exactly one owner, the data associated with the cell need only be stored once. Scientific datasets may contain a large number of attributes for each point, so this can result in significant space savings over a full duplication method.

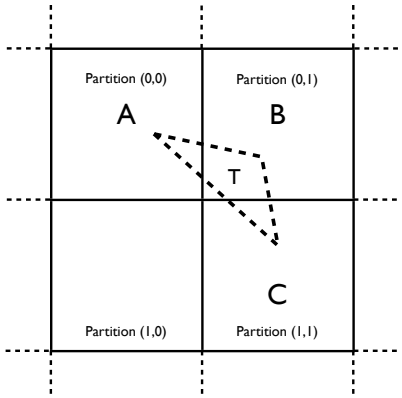

Figure 4.3. Handling triangles that span boundaries.

Looking at figure 4.3, we see the triangular cell T overlaps cell groups A, B and C. The index space occupied by cell group A will have the smallest coordinates in each dimension (compared to B and C), and is therefore chosen to be the owner of the cell. All the information pertaining to T, including the cell, vertex, and data values will be recorded in cell group A. Subsequently, when T is represented in B and C, it is recorded as a borrowed

cell, so these cell groups will refer to A for complete information about T.

4.4 Efficient Representation of Boundary Cells

We address the boundary cells that span the partition lines by creating a scheme that avoids unnecessary duplication. We call this scheme the *owned and borrowed scheme*.

A set $\mathbb{S}$ of simplices occur arbitrarily in the dataset domain D in an unstructured dataset. An n-dimensional partitioning of D will produce partitions P_i such that

$$\bigcup_{i=1}^{N} P_i = D \qquad \text{and} \qquad \bigcap_{i=1}^{N} P_i = \emptyset.$$

Since the data is unstructured, we cannot predetermine or control the number of simplices that span a partition boundary. If S be the set of simplices that span partition boundaries such that $S \subseteq \mathbb{S}$. Our goal is to avoid the unnecessary duplication of S across multiple partitions.

The owner borrower scheme is presented in the following algorithm:

input : P: Partitions, C: list of simplicial cells

Result: O: List of owned cells, B: List of borrowed cells

for *each cell* c_i *in* C **do**
  P_b = map bounding box of c_i to partitioning grid
  for *each partition element* $p_i \in P_b$ **do**
    if c_i.*containedBy*(p_i) **then**
      c_i.owner = p_i;
      c_i.hasOwner = true;
      add c_i to O;
    else if c_i.*intersects*(p_i) **then**
      if c_i.*hasOwner* = *false* **then**
        c_i.owner = p_i;
        c_i.hasOwner = true;
        add c_i to O;
      else
        add c_i to B;
      end
  end
end

Algorithm 1: Owner–borrower scheme

If Δ is the extra storage space we use when we split the data, in the typical case, Δ is much smaller than the amount of space that would be required to store border simplices, if the border simplices are duplicated across the multiple partitions.

4.5 In Memory Subgrids

A *subgrid* is a further partitioning of each partition element stored on disk and is used to further accelerate search for cells when they are read into memory. The datum query can be more quickly resolved due to the much smaller number of triangles intersecting the relevant subgrid partition element. We perform a more thorogh investigation of the subgrid and how it affects performance in chapter 5.

4.6 Storage Model

After the splitting process is complete, we require an efficient mechanism for storing the data to facilitate fast retrieval. Spatial datasets must ultimately be represented as one dimensional files on disk or a similar storage device. A *storage model* describes how the n-dimensional domain of the dataset is mapped to the one dimensional file space [68].

4.6.1 The Rod Storage Model

Some file formats can be described using the *rod storage model*, which views files as a collection of *rods*, where each rod is a sequence of elements that are contiguous in both the n-dimensional data space and the one dimensional file space [56]. For example, the rasters of a raw image file can be considered rods if the image file is stored in the usual *linear* order.

Figure 4.4 shows the rod storage model. Since the elements of a rod are contiguous on disk, they can be accessed with a single read operation. When fulfilling a subblock query, the query region can be broken into a collection of rod subsets, which each conceptually represent a single read operation.

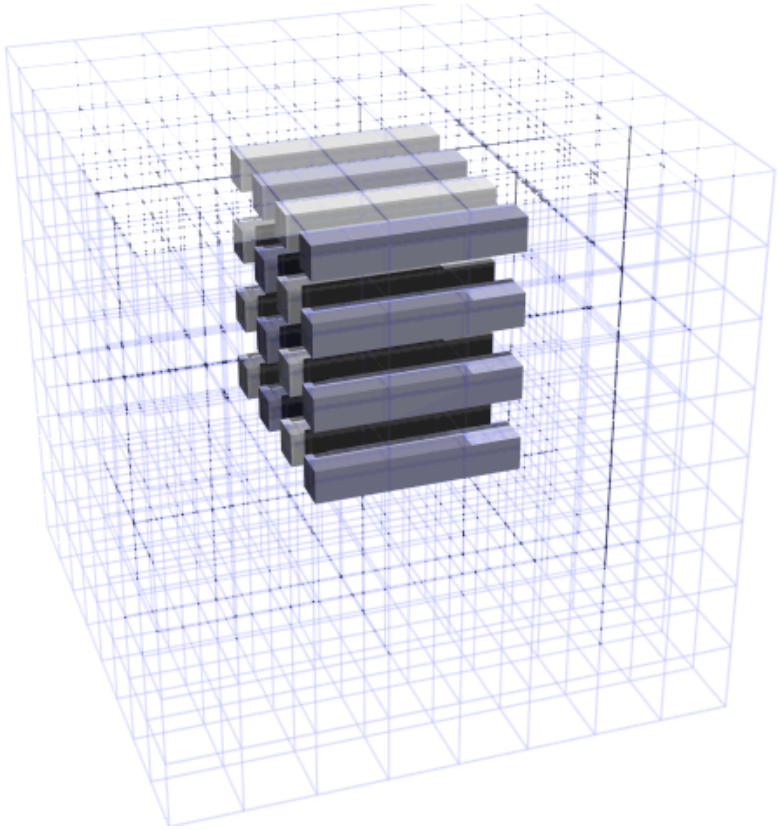

Figure 4.4. Rod Storage Model

4.6.2 Merged File Format

Our previous work [56] applies the IAP caching mechanism to files that fit the rod storage model, including both linear and chunked [63] files. The major motivation of the work described here is to fit unstructured grids to the rod storage model, which allows a series of adjacent elements to be read from file in a single read transaction. In doing so, we can once again achieve very significant gains in performance by taking advantage of prior knowledge of the access pattern. Merging the data into a single file helps us take advantage of *filesystem prefetching*[65], which is only effective when reading from a single file. Merging the cell groups into a single file also improves the locality of reference in the one dimensional file space on disk. For these reasons, we merge the cell groups created from the splitting phase back into a single file.

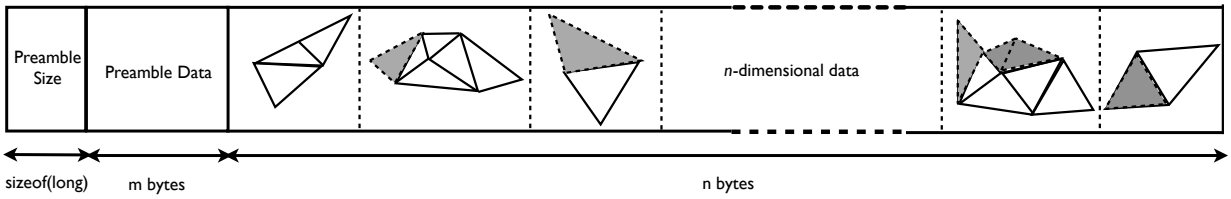

Figure 4.5. Resulting file after merging the partition elements

We created a merged file format with a preamble that maintains metadata about the cell groups stored in the data section using the rod storage model. The merged file is as shown in figure 4.5. The preamble can be read into memory providing the Granite system with a description of the complete dataset. A future possibility during the merging process may be to use a space filling curve for merging the cell groups into a single one dimensional file.

The implication of the owner borrower scheme and the merged file is presented. Let $D(f)$ be the function that produces the data corresponding to f. The data storage space occupied by the resulting merged file M is given by equation 4.1

$$D(M) = D(\mathbb{S}) + D(S) + D(H) \tag{4.1}$$

where $\mathbb{S}$ is the set of simplices that occur arbitrarily in the dataset domain, S is the set of simplices that span partition boundaries such that $S \subseteq \mathbb{S}$, and H is the header and preamble information. The extra storage space Δ when we use the owner borrower scheme is given by equation 4.2.

$$\Delta = D(M) - D(\mathbb{S}) = D(S) + D(H) \tag{4.2}$$

Our results so far show that very significant improvements can be derived when the data associated with each vertex is large. In this case, Δ is negligible. This is because we are able to improve performance tremendously due to our partitioning mechanism and the owner borrower scheme. In addition, our scheme reduces the vast amount of hopping around on disk that would have otherwise occurred.

4.6.2.1 Preamble Information

The preamble contains the metadata for the merged file. It stores information for each cell group contained in the merged file. The preamble contains the following information per cell group:

1. *Offset*: a long integer value written into the preamble indicating the file's offset within the merged file.

2. *Size*: stores information about the file length.

3. *Dimension*: stores information about the dimensionality of each file.

4. *Record Size*: writes the size in integer for each record in the file.

5. *Number of Fields*: Writes the number of fields per record in the file.

4.6.2.2 Merged n-Dimensional Data

The Index Space iterator (*ISIterator*) specifies an integer index space containing grid coordinates that contains the merged data. The user specifies the directory of the files to be merged and an output directory and file name of the data to be merged. For example, an ISIterator with a 2x2 dimension will have grids [0,0], [0,1], [1,0] and [1,1]. There are 6 split files that make up each partition element. They are:

1. Attribute file (.attr): stores point informatiom
2. Borrowed sub grid file (.borrowed.subgrid): stores the borrowed in-memory cells
3. Owned sub grid file (.owned.subgrid): stores the owned in-memory cells
4. The owned triangle (.tri): stores owned cell information
5. The borrowed triangle (.borrowed.tri): stores borrowed cell information
6. The data file (.data): stores the n–dimensional data

Each of the files above has an associated descriptor file in *xfdl* format.

4.6.2.3 XFDL File Format

The extensible file descriptor language (xfdl) file format was created for the purpose of describing the data set and can be used to represent both structured and unstructured multidimensional scientific data. It contains information about the accompanying data such as dimensionality, number of fields, the record size, range of information, in an XML format.

In this case however, the Granite system will merge the existing data (the split files created), into a single physical data file using information available in the accompanying XFDL files.

CHAPTER 5

IN MEMORY PARTITIONING

Splitting or chunking is a popular approach used to handle large sized data that will not fit into a single machines memory. Several partitioning techniques [30, 38, 58, 76, 64] have been thoroughly researched. However, most of the available techniques are not directly applicable to unstructured grids due to the intricate relationship that exists between the sample points of the data and the vertices corresponding to a cell can be anywhere in the vertex file. Some past work [52, 29] related to unstructured grids have been proposed that use edge walking techniques to partition unstructured meshes. Most of these techniques do not make provision for the underlying data associated with the vertices and therefore results in duplication of data across partition boundaries.

We have investigated different partitionings on disk using partitioning configurations that creates partition elements similar to the one depicted in figure 5.1*a* and 5.1*b*. Other partition element possibilities are shown in figure 5.2. We have analyzed their overall performance when we access the data using iteration aware prefetching and when we do not. The various partitionings have associated advantages and disadvantages depending on the access pattern, and has been investigated. From our results, the cubic partitioning configuration that creates a cubic shaped partition element similar to figure 5.2*a*, have a relatively average performance across multiple access patterns when compared to the other partition elements.

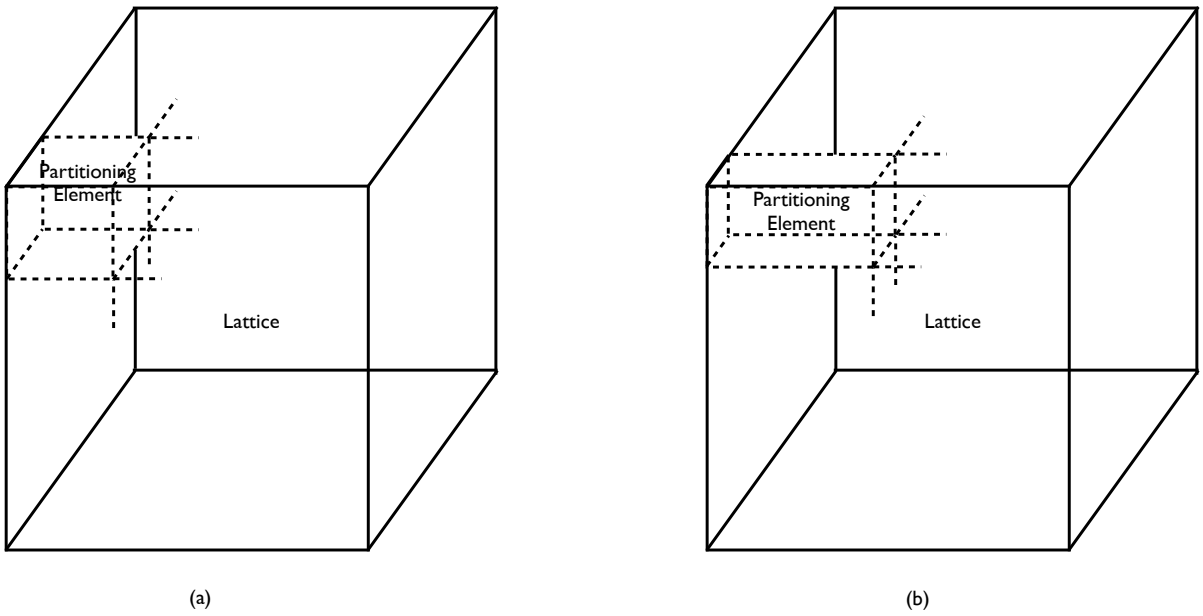

Figure 5.1. Configuration used in partitioning a Lattice determines the shape of the resulting partition element

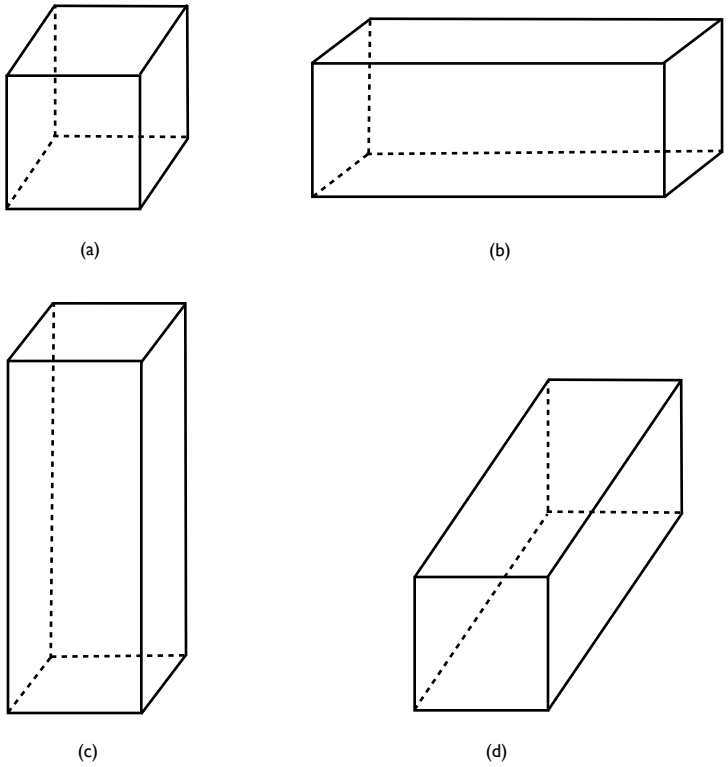

Figure 5.2. 3D In-Memory Partition Elements Shapes Possibilities

In order to further improve access performance, when a partition element is loaded into memory, we perform a further partitioning of the partition element in memory. We term the partitioning performed in memory as the *in-memory partitioning*, and the partitioning on disk as the *disk level partitioning*.

Varying the disk and in-memory partitioning configurations will only create shapes similar to the ones depicted in figure 5.2, or with slight variations. We gained considerable insights about the relationship between disk level partitioning and how it affects performance and present our results in chapter 9. We perform extensive tests to gain further insight into in-memory partitioning by keeping the disk level partition the same, while varying the in-memory partitioning and analyze the effect on the overall system performance.

If prior knowledge of how the data will be accessed is available, we can recommend an efficient disk level partitioning that will favor the user specified access pattern. A smaller shaped partition element is light weight due to its relative size on disk. It has associated with it characteristics such as low bandwidth and high latency costs due to the frequent loading and reloading of partition elements during data access. Bigger shaped partition elements are heavy weight due to their high bandwidth. However, they have lower overall latency costs because the loading and reloading is performed less frequently.

During data access when a queried cell is not currently in memory, the missing owner partition needs to be read from disk into memory. After the owner partition has been read, the data and other information associated with the cell is available and can be used as required for interpolation or other tasks.

5.1 In-Memory Partitioning

Fast access to data is a primary goal of storage systems. When unstructured spatial datasets are queried, the query point often does not correspond to an actual vertex in the dataset. Cells are therefore essential for interpolating the values corresponding to a query point. Our

partitioning mechanism significantly reduces the number of containment tests that need to be performed in order to identify the cell containing a query point.

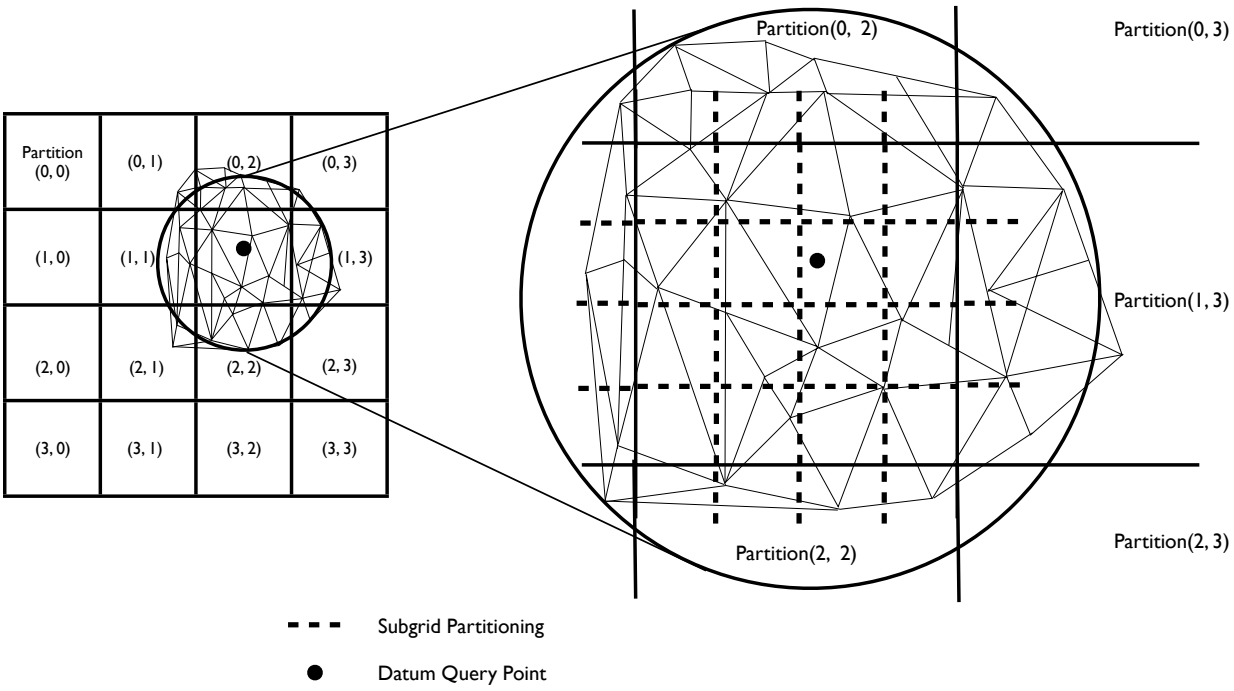

Figure 5.3. A datum query showing *subgrid* partitioning within a partition element.

Consider the grid partitioning in figure 5.3 showing a 3×3 disk level partitioning that creates 16 partition elements. Suppose a user queries the lattice representation of the dataset and the query falls on partition (1,2) as shown in figure 5.3. The partitioning essentially eliminates 15 partition elements, along with the bandwidth and latency costs associated with each partition element that we do not have to consider in resolving the query point. We can then load the cell group corresponding to partition (1,2) into memory and perform containment tests.

The in-memory partitioning is performed to further speed up the process of identifying the cell containing or intersecting with a query point. This process significantly reduces the number of candidate cells that needs to be tested for containment. In chapter 4, we focused more on how our partitioning mechanism affected performance at the disk level. In this

chapter, we investigate and analyze the relationship between the in-memory partitioning and disk level partitioning, and the overall effect on performance.

When the data is present in memory, we are less concerned about the bandwidth and latency associated with loading partition elements like we are at the disk level, and more concerned about what effects the granularity of the in-memory partitioning will have on data retrieval and the overall system performance. A fine grained partitioning will create more boundary cell cases across partition lines and also leads to an increase in the overall data size, while a coarse partitioning will have fewer borrowed cells across multiple partitions as shown in figure 5.4. Our owner–borrower scheme is also used in memory and significantly reduces duplication across multiple partition boundaries. Our goal is to balance the partitioning and reduce the number of borrowed cells across partition boundary such that we are able to efficiently reduce the number of candidate cells needed for containment tests. We present our results in chapter 9.

We significantly improve the retrieval performance by keeping in memory the cell that contained the previous query and testing it first. This minor optimization produces significant gains in performance because the number of containment tests that would have been performed is further reduced. When query points are close to each other as in a fine grained iteration, more query points intersect with the same cell and by keeping the previous cell in memory, we eliminate the costs of locating and reading that cell from disk.

In figure 5.4*a*, the partitioning is more coarse and the locality provided by the partitioning is not very effective as there is still a considerable number of candidate cells per partition element. Figure 5.4*d* has very fine grained partitioning and provides very high locality. Each partition is however too fine grained as the number of cells that span the partition boundary increases considerably. The algorithm is able to quickly determine the containment test but performance suffers as there are more borrowed cells and retrieving the data from the owner partition degrades the overall system performance. Figures 5.4*b* and 5.4*c* are a tradeoff between both extremes. Deciding the best partitioning and identifying a

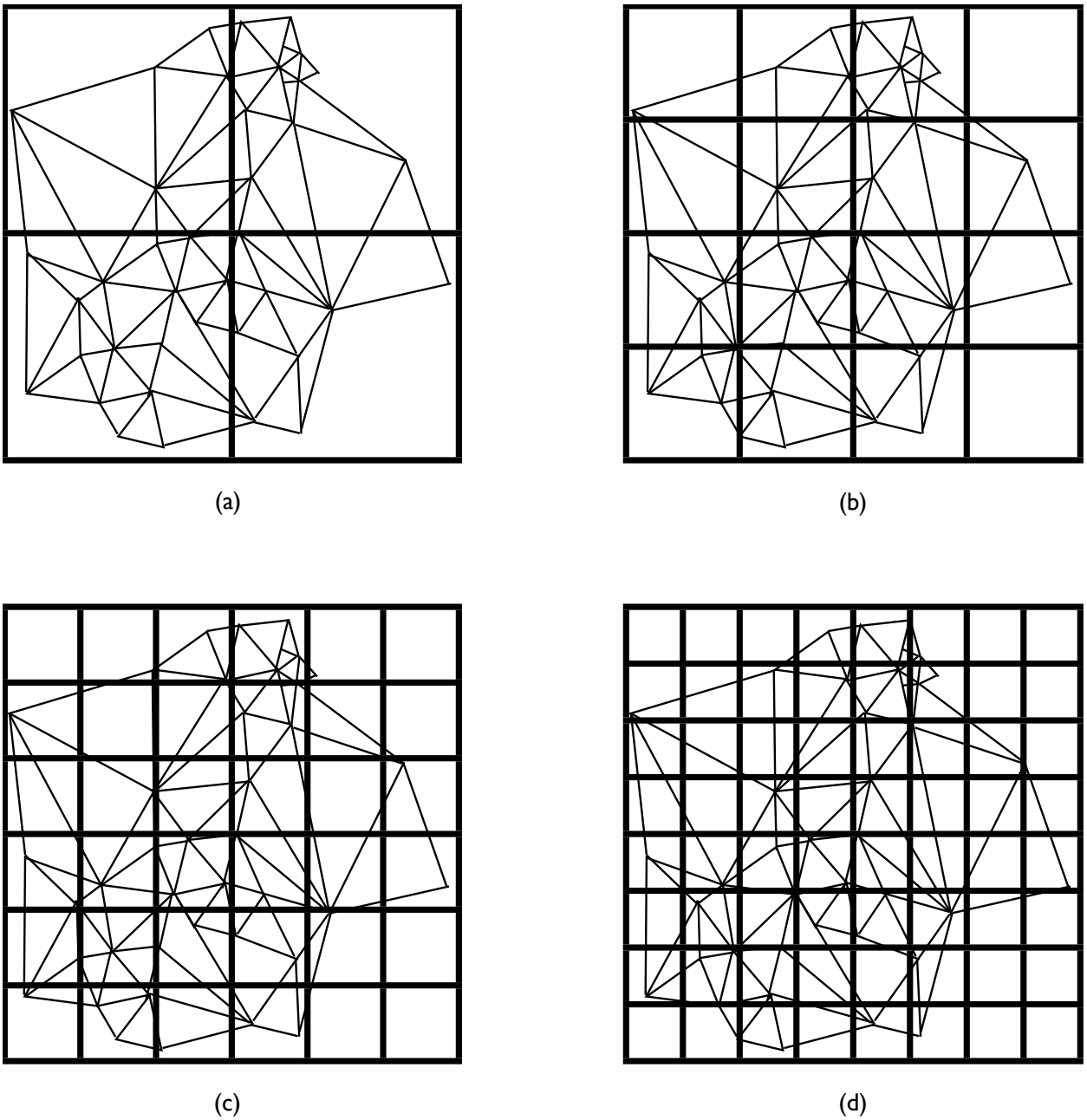

Figure 5.4. Granularity of In-Memory Partitioning

"sweet spot" is a major motivation and focus of our testing. Using our synthetic data that has vertices spread out relatively evenly across the dataset, we are able to determine an effective in-memory partitioning for a cubic dataset. This is verified by running the same tests using real data with uneven vertex distribution and achieved similar results. The results are presented in chapter 9.

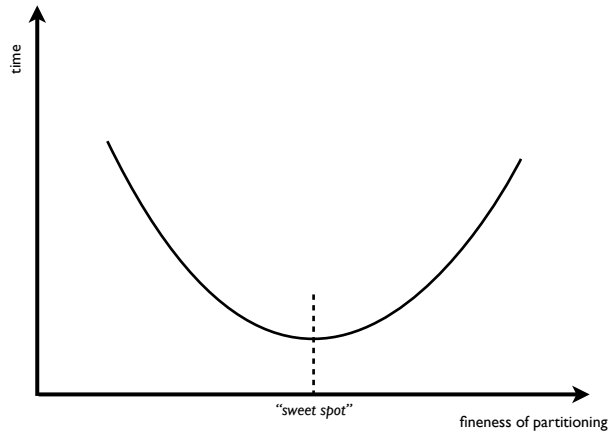

Figure 5.5. Granularity of in-memory partitioning

Providing a way to choose the coarseness of the resulting grid after partitioning, for our purposes, a partitioning $\mathbb{P}$ can be described as the regular tessellation of a domain $\mathbb{P}_D$ into pieces (elements) formed by dividing each axis i into $\mathbb{P}_{C_i}$ equal intervals. That is, the length d_i of each element in axis i is simply

$$d_i = \frac{\mathbb{P}_{D_i}}{\mathbb{P}_{C_i}}, i \in 0...n-1 \tag{5.1}$$

where $\mathbb{P}_{D_i}$ is the length of the domain axes and $\mathbb{P}_{C_i}$ is the number of elements along axis i.

Assuming a uniform distribution of cells across the dataset domain, the disk level partitioning $\mathbb{S}$ divides the set of N cells into groups, where each group corresponds to a partition element, and has G cells:

$$G = \frac{N}{|\mathbb{S}|} \tag{5.2}$$

where $|\mathbb{S}|$ is defined to be the number of elements in the partitioning $\mathbb{S}$.

It remains to choose $\mathbb{M}$, the in-memory partitioning. We select $\mathbb{M}_{C_i}$ such that:

$$\frac{G}{|\mathbb{M}|} = n \tag{5.3}$$

where n is the number of simplices per element of $\mathbb{M}$. The ideal choice for n depends on the characteristics of the hardware. However, we present experimental observations for this important parameter in our results in chapter 9.

CHAPTER 6

CACHING AND PREFETCHING

6.1 Prefetching

Prefetching has long been used to speed up execution both at the system and application level. Over the years, the computing community have tried to improve system throughput by integrating prefetching into hardware and software systems [23, 16]. The filesystem cache also prefetches pages following an explicitly accessed page in the hope that the prefetched page will be accessed next and reads to disk will be reduced. However, filesystem prefetching will increase the number of inappropriate pages loaded when the user does not proceed through the file in the manner the filesystem predicts, which degrades performance. Our application level caching mechanism reconciles the user access pattern with filesystem prefetching, allowing it to work much more effectively to boost performance.

Our approach offers two means by which data can be prefetched. In the first case, we apply Iteration Aware Prefetching (IAP) to the access method when advance knowledge of the user's access pattern is known and data can be prefetched into a cache that maintains a spatial view of the data. Otherwise, we use a simple Least Recently Used (LRU) cache when we do not have advance knowledge of the access pattern. Performance is best when we are able to apply IAP than when we are not able to, because we can conceptually read

a sequence of partition elements from disk into memory in a single read transaction.

6.2 Least Recently Used (LRU) Cache (No IAP)

When the user specifies an iterator pattern, the granularity can be easily obtained from the iterator. We have established that the configuration of the partitioning will affect the overall performance. When prior knowledge of the user access pattern is unknown, we require some other mechanism for loading partition elements from disk into memory. We use a simple LRU cache to achieve this. The LRU cache uses n-dimensional cache blocks and maintains the spatial nature of the data.

Due to the latency and bandwidth costs associated with loading and reloading partition elements from the LRU cache, performance is best when the size of the LRU cache conveniently contains all the partition elements that will be accessed in a run axis. Thus, if an iteration continues within the same rod, we do not have to reload partition elements that are held in the LRU cache.

6.3 Iteration Aware Prefetching

The speed of data retrieval from disk suffers when data is accessed in a manner different from how the data is stored. The underlying file structure is one dimensional and data is stored (mostly) contiguously on disk. If the physical file is accessed sequentially, the filesystem cache performs quite well. When the data is not accessed sequentially, performance degrades quickly. Several reads may be required for portions of a file that are neighbors in the data space but not in the one dimensional file space.

Iterator Aware Prefetching is a prefetching approach applicable to situations in which the access pattern is not data dependent and can be known in advance. An iterator describes the access pattern and also performs the iteration through the data space. We use a separate thread for reading partition elements into the IAP cache. Because of this prior knowledge

provided by the iterator, we can configure a cache that uses n-dimensional cache blocks with a shape tuned to the iteration. Because of this block shape, data is never loaded more than once into an IAP cache, assuming the iterator is used to determine the access pattern [56].

6.4 Out-of-Core Prefetching

The lattice layer, like the datasource layer, uses an *out-of-core* approach to data access, loading only necessary subsets of the larger data volume. This capability is made possible by the partitioning and owned and borrowed cell information. Because of this, we can greatly reduce the amount of memory needed at one time by only prefetching cell groups that will actually be used in satisfying user query. Available memory does not limit the size of the datasets that our system can handle.

6.5 Access Pattern and Cache Blocks

When we create a cache block for use with unstructured grids, we use one or more *slices* of the iteration space. A slice is formed by cutting the space with a plane orthogonal to a specified *slice axis*. The dimensionality of the resulting slice matches the dimensionality of the dataset, but has a thickness of one in the slice dimension.

The algorithm for for constructing cache slices is presented in algorithm 2. This algorithm is a simplification of the IAP algorithm presented in our past work [56] but each slice contains multiple data values.

Algorithm A:

Input:

$Lower = \{l_0, l_1, l_2, ..., l_{n-1}\}$: Lower corner of iteration space

$Upper = \{u_0, u_1, u_2, ..., u_{n-1}\}$: Upper corner of iteration space

$A_n = \{a_0, a_1, a_2, ..., a_{n-1}\}$: Iterator Ordering

Output:

$S = \{s_0, s_1, s_2, ..., s_{n-1}\}$ that represents the shape of a slice of the iteration space that is tuned to the iteration ordering

begin

for $i = 0$ **up to** $n - 1$

if i != a_0 *// most significant axis*

$s_i = Upper_i - Lower_i$

else

$s_i = 1$

end

return S

Algorithm 2: Algorithm A produces cache block slices tuned to the given iteration

In order to create a cache block shape that performs well when used with a particular iterator, we choose a slice axis that is also the most significant axis of the axis ordering associated with the iterator (see section 3.3.6). The position of the iterator is monotonically increasing on this axis, so the iterator is certain never to revisit a slice once it has left it.

Data in a cache block is therefore never loaded more than once.

Such a scheme cannot work with unstructured grids unless we can quickly and efficiently load the cells corresponding to a slice of the iteration space. Because the merged data format described in section 4.6.2 makes this possible, we can work with the index space formed by the lattice partitioning instead of slicing the geometric data space. That is, cache blocks will consist of collections of partition elements, each containing a cell group and associated data. Better yet, whole sequences of partition elements can be read with a single transaction, because the rod storage model applies to the merged data format.

After the first row of elements is loaded into the cache, an iterator's data requirements would be satisfied from the cache for several rows of the iteration, assuming the spacing of the iterator's rows is much smaller than the height of a partition element. Eventually, a new row of the iteration will be started which falls outside of the first row of partition elements. This triggers a cache miss, which causes the second row of elements to be loaded into the cache. The first row will not be needed again during the iteration, which will proceed in similar fashion until complete.

Behavior in three dimensional datasets is analogous, except the cache will contain whole slices (i.e. planes) of data, rather than just rows.

6.6 Group List

Whenever we do not use an IAP cache, we require some other mechanism for loading and maintaining cell groups in memory to improve the data retrieval performance. We introduce the concept of *group lists*.

The group list is an LRU cache of cell groups used to satisfy user queries. The number of cell groups loaded must not exceed available memory, but must be large enough to effectively improve performance.

Consider the diagram shown in figure 6.1. In figure 6.1*a*, the size of the group list

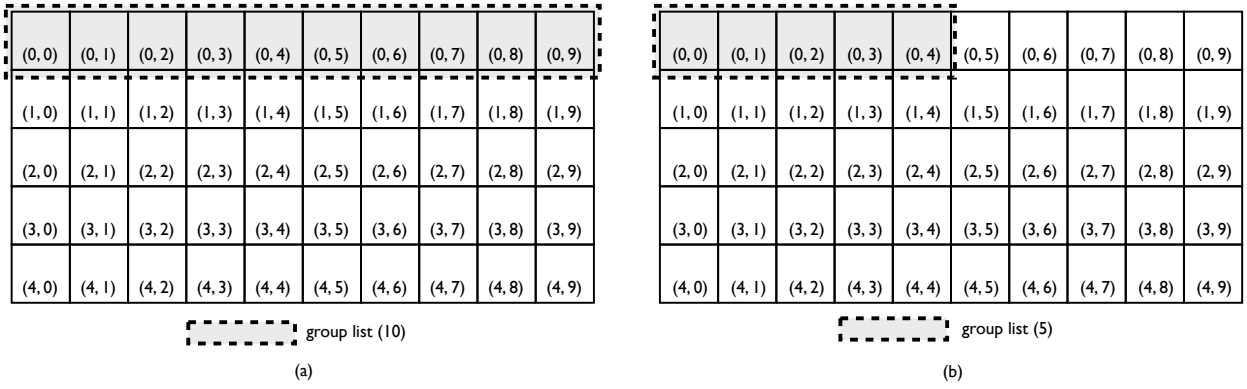

Figure 6.1. 2D dataset showing relationship between the size of the group lists and the rod storage model. (a) Rod length is 10 and the group list is 10 (b) Rod length is 10 and the group list is 5

(i.e. the number of cell groups it can hold) corresponds to the length of the rod spanning the iteration space. In figure 6.1*b* however, the size of the group list is smaller, and does not extend through the length of the storage rod. When the application queries data not contained by the group list, a new cell group that satisfies the query will be loaded from disk into the list, and the least recently used group will be ejected from the list.

When the LRU cache is not large enough, this behavior could potentially hurt performance when iterating in row by row fashion through the data space, because the recently ejected group may be required when the iteration returns to the left side. The frequency with which this occurs is related to the length of the group list as well as the granularity of the iteration.

We perform tests using the 3D dataset we describe in section 9.1.3. The dataset is approximately 2GB. We use the partitioning configurations 10x10x10, 10x18x5, 18x10x5, and 10x10x20. These partitioning configurations create partition element shapes similar to figure 3.10. We vary the group list size to hold between 5 and 240 cell groups. By varying the group list size, the size will be less than, equal to and greater than the rod length within each partitioning configuration. The actual size in bytes of the group list varies for each partitioning configuration but can be estimated. For example in the 10x10x10 partitioning,

assuming uniform distribution of data points in the domain, the relative size of each partition element is approximately 2MB. The group list size in bytes is thus 2MB multiplied by the number of cell groups.

We present the results in figures 6.2 and 6.3. Figure 6.2 shows the results using the merged file and figure 6.3 shows the results using the reorganized files. The graphs show the effect the size of group lists have on the overall performance. In all cases, there is a dramatic improvement in performance when the group list is large enough to span the iteration space. Because the larger group list reduces the frequency of group reloading, the overall access time drops dramatically.

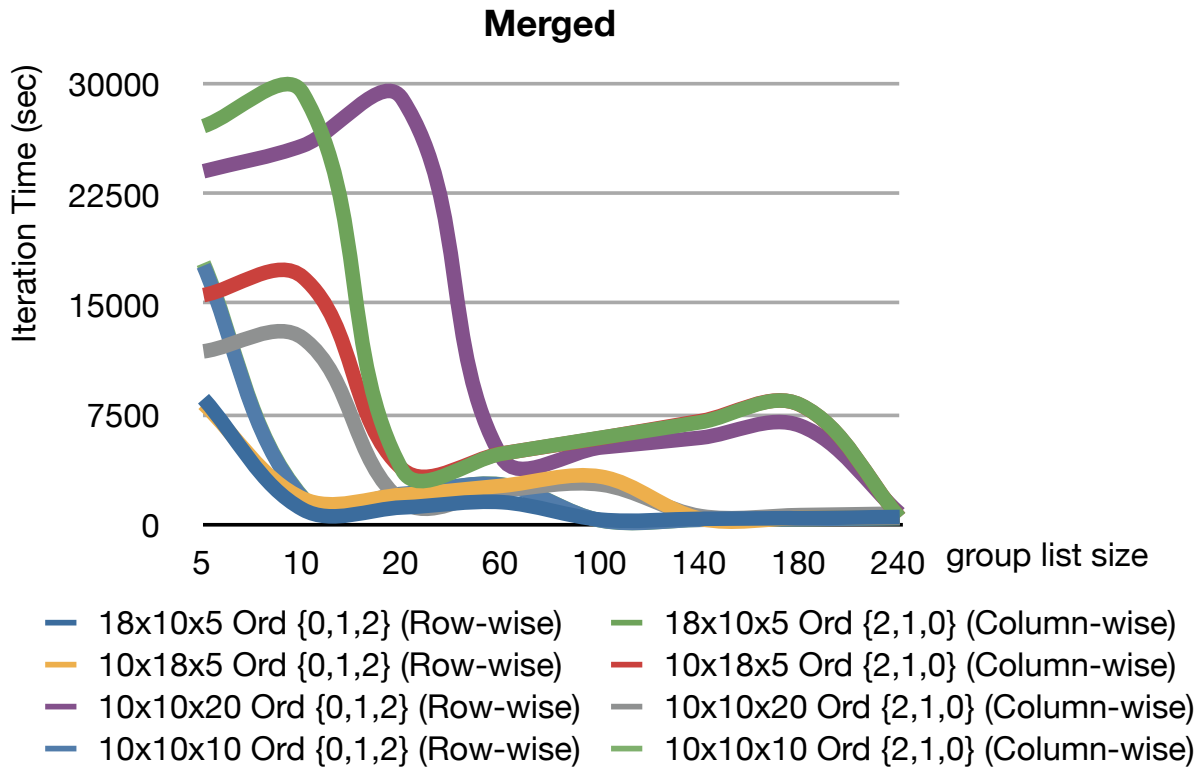

Figure 6.2. Effect of varying group list length on the merged file. The size of the LRU cache is determined by the number of cell groups it can hold. In all cases, there is a dramatic improvement in performance when the group list is large enough to span the iteration space

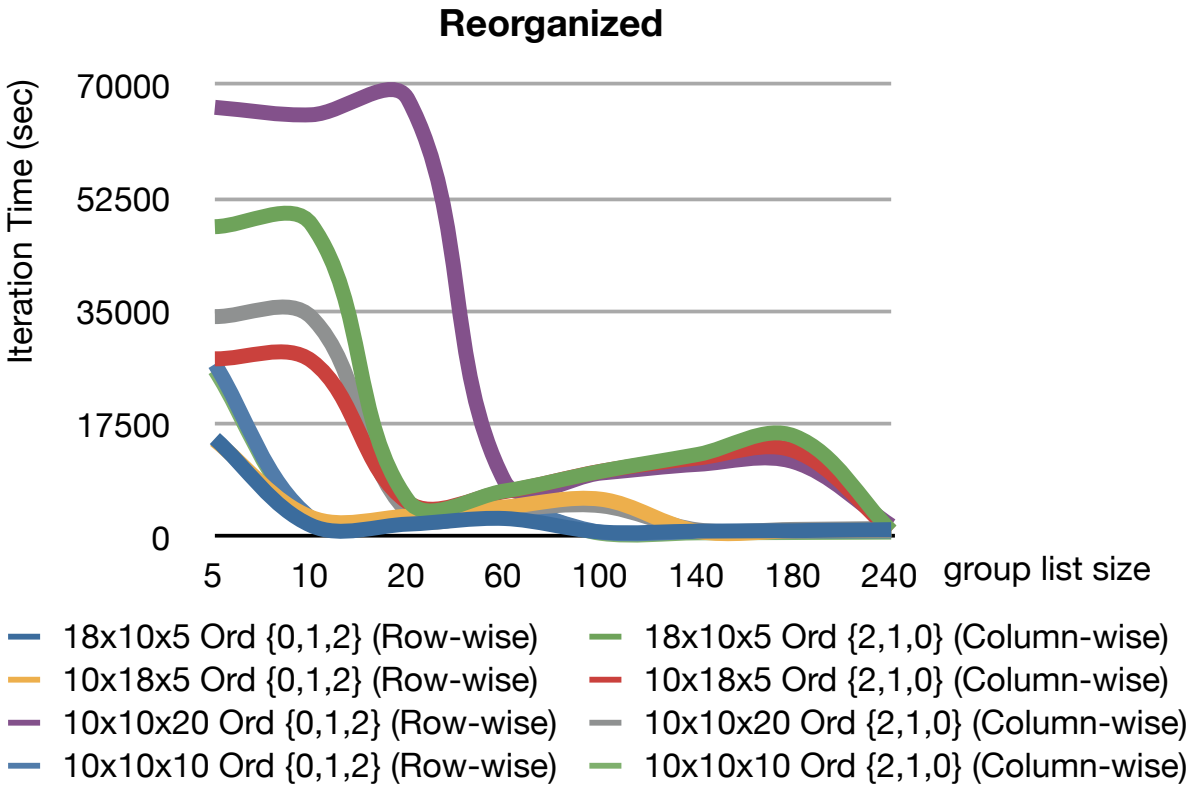

Figure 6.3. Effect of varying group list length on the reorganized files. The size of the LRU cache is determined by the number of cell groups it can hold. In all cases, there is a dramatic improvement in performance when the group list is large enough to span the iteration space

6.7 Granularity of Iteration

Larger partitions incur larger bandwidth costs. When choosing the size of the partition elements for our file format, a range of access patterns should be considered. At one extreme, completely random access may use only a single datum from a partition element before it is discarded. In this case, smaller partition elements reduce the volume of unused data, improving performance.

At the other end of the range, a finely (densely) spaced iteration may use most or all of the data loaded as part of the element. In this case, the cost of reading the larger data volume of big partition elements is worth paying. The data is being used before being discarded, and we are also reducing the total number of read operations required by the access pattern.

Between these two extremes, we have coarser iterations that skip some of the elements in the partitioning, as shown in figure 6.4. If the partitioning is sufficiently fine grained compared to the iteration, then we may avoid loading at least some unneeded data. Figure 6.4 shows the same iteration applied to two different partitionings. The partitioning used in figure 6.4*a* will be better suited for a finer iteration that more heavily reuses partition elements. In figure 6.4*b*, we skip entire rods of unneeded elements due solely to the partition shape.

The image shown in figure 6.5 shows the effect of a coarse iteration step of 0.01. This is a different image, made with the same data as the one depicted in figure 3.7 which uses a finer grained step of 0.001. The level of detail is very much reduced in figure 6.5.

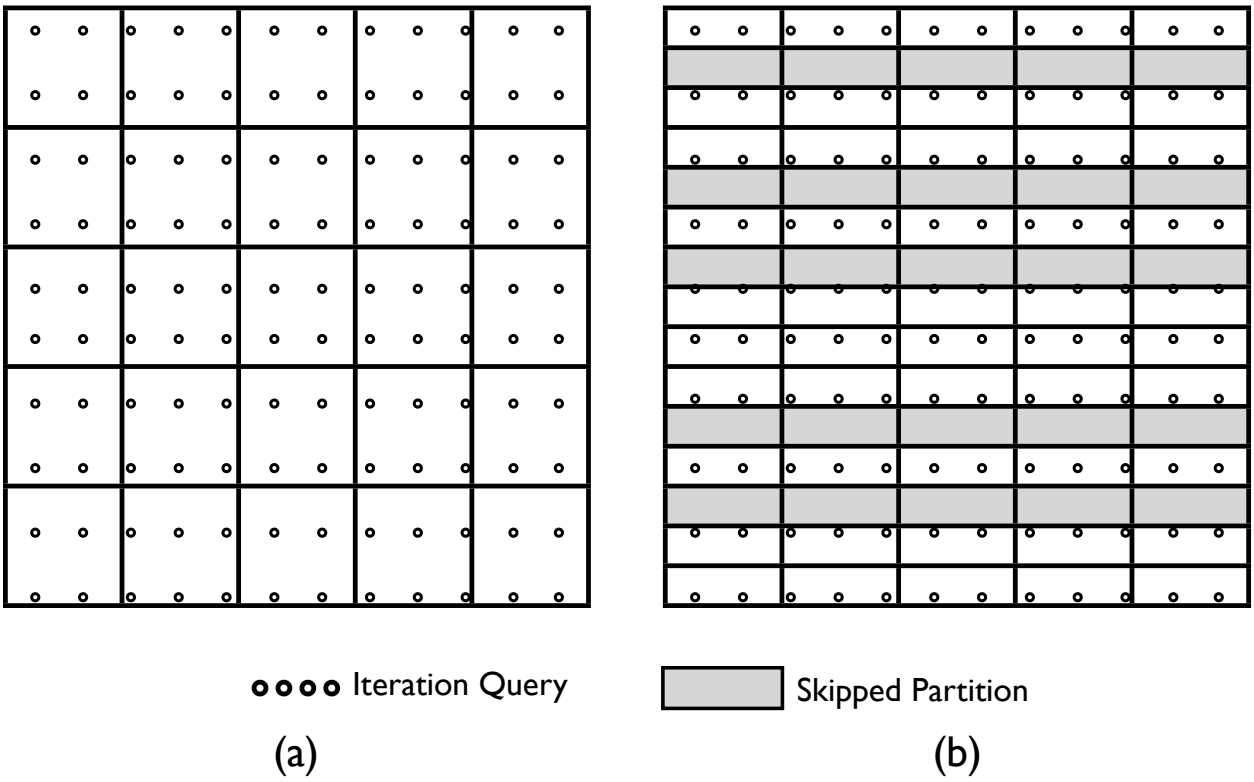

Figure 6.4. Skipped partitions using similar iterations on different partitionings

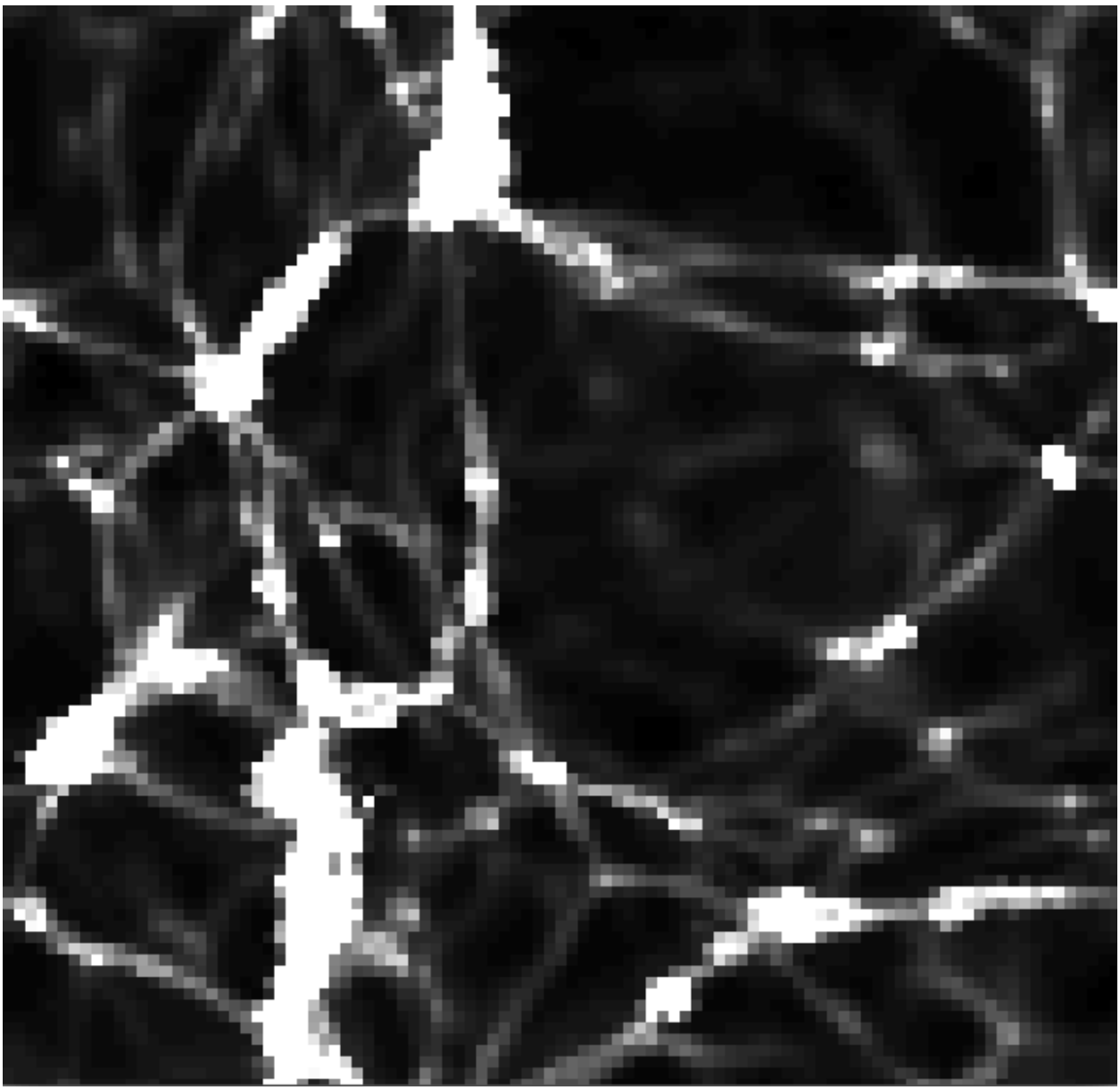

Figure 6.5. Effect of using a coarse iteration step on visualization. Internal energy in the Arepo data at z=0.5

CHAPTER 7

DECLUSTERING

Our partitioning mechanism effectively partitions unstructured grids into disjoint regions with overlaps handled by the owner borrower scheme. We go one step further and apply declustering to the dataset in order to take advantage of I/O parallelism. Declustering is a mechanism that distributes data across multiple disks in order to reduce the overall retrieval costs. Data declustering is sometimes used to maintain duplicate data in order to perform different tasks across the data [70]. In this work, we apply declustering to take advantage of the I/O parallelism and analyze the performance of our approach. IAP is applied to the declustering mechanism for data access. IAP forms block queries and in this case is broken up across the disks. We term this case *parallel IAP*.

Several declustering mechanisms have been proposed [26], [37], [50] and recent research has focused on declustering using replication [13] [54], [70]. Declustering raises a few concerns caused by skew as a result of the distribution of data within the dataset domain and how the data is processed. The first is the distribution skew which is more concerned with load balancing among partitions. Our partitioning mechanism is capable of creating coarse or fine grained partitions and the bandwidth costs varies due to the size of the created partition element. The second is the processing skew. This tries to ensure that a disk is not over worked while another available disk is idle. We focus our work in this section

on improving access performance to unstructured grids. We do not propose a new scheme. However, we successfully apply an existing declustering mechanism to unstructured grids which to our knowledge has not been done.

When the disk level partitioning is very fine grained, our goal is to improve the overall performance time by amortizing latency costs over n partition elements, instead of paying similar costs for each individual partition element, where n is the number of disks we access in parallel.

A declustering mechanism is required that will reduce the distribution and processing skew among multiple disks. The disk modulo (DM) [26] is a popular spatial declustering technique that assigns partitions (buckets) to disks in a manner that achieves maximum disk access concurrency for partial queries thereby minimizing response time. The choice of DM to our approach is based upon analysis of its effectiveness, intuitiveness and yet simple approach, and its applicability to grid files. It also does not place a restriction on the attribute dimensionality or the number of disks that may be used, which is a good fit for our purposes. DM has been shown to be optimal under many conditions that occur in real world applications [24] and perform well for relatively small number of disks.

7.1 Disk Modulo

Figure 7.1 depicts the DM approach but in our case, each partition element corresponds to buckets that are stored on multiple disks. DM is highly adaptable to region queries [75, 22, 48] as it ensures that a single disk is not overworked when reading from a spatial region since the partition elements are effectively spread out across multiple disks. This presents an avenue for future work in our declustered approach. We hope to investigate alternative approaches to assigning partition elements to disk such as space feeling curves [61, 34] and evaluate the performance.

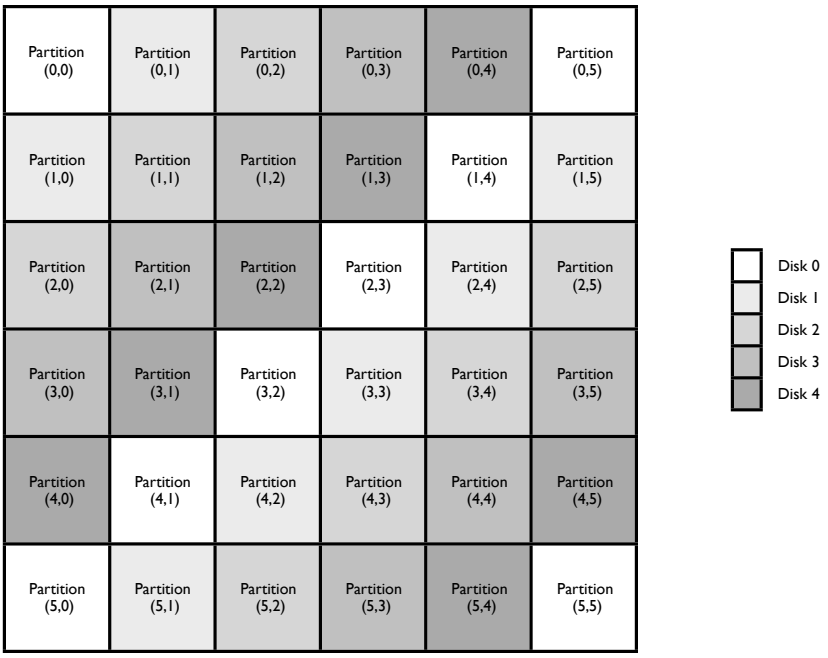

Figure 7.1. Mapping of partitioning elemenst to disks using Disk Modulo (DM)

7.2 Parallel IAP

IAP provides advanced knowledge of the user's access pattern and creates cache blocks with a shape tuned to the iteration. In the declustering approach, during data access, we prefetch data in parallel into a spatial cache using information provided by IAP. We are able to read multiple partition elements from disk into memory and pay retrieval cost for a single read while in fact performing multiple reads. The pseudocode used in performing parallel IAP is presented in figure 7.2

```
IndexSpaceID[] theBounds = new IntegerIndexSpaceID[iapCacheBounds.volume()];

// cell groups to be accessed in parallel
FileArrayCellGroup[] theGroups;

int amt, amtGreaterThanIndex, amtNeeded;
for(i=0; i<theBounds.length; i+=NUMDISK) {

        // map each index space to a cell group
        for(k=0; k<NUMDISK; k++) {
        theGroups[k]= mapToCellGroup(theBounds[i+k]);
    }

    Thread[] t = new Thread[]{theGroups};

    // start all threads
        t.start();

    // wait for threads to finish reading before adding to group list
    try {
        for (Thread ts : new Thread[] {new Thread(theGroups})
            for (Thread ts : new Thread[] {t})
                        ts.join();
                } catch (InterruptedException e){
                        System.out.println("Exception in parallel IAP");
                }

    //insert cell groups into grouplist
        groupList.insert(theGroups);
}

return theGroup;
```

Figure 7.2. Prefetching partition elements from multiple disks in parallel

CHAPTER 8

VISUALIZATION

One of the many areas our approach may be utilized is in visualization. Slice visualization [25] has been around for a while and is used to gain insight into spatial data. The *Slicer* application [57] for example, is capable of displaying progressive two dimensional *slice planes* of a three dimensional volume.

We wrote a visualizer for the lattice that iterates through the data space querying the lattice for information that corresponds to the query point. The images shown in figures 3.7 and 8.1 are generated from the *Arepo* dataset [46] using the Granite systems visualization capabilities. Figure 3.7 shows energy distribution in the lattice at z=0.5. The image in figure 8.1 shows an image generated when we use the declustering mechanism to access real unstructured grids. The image shows energy distribution in the lattice at z=0.3.

We create a lattice from the dataset and use IAP for iterating through the dataset and map the data value to a color. Sample code for creating the lattice and the visualization within the Granite system is shown in figure 8.2.

8.1 Determining cell intersection

Several approaches are available for determining if a point falls inside a tetrahedron [8]. In our current implementation, we evaluate plane equations for each of the four facets at the

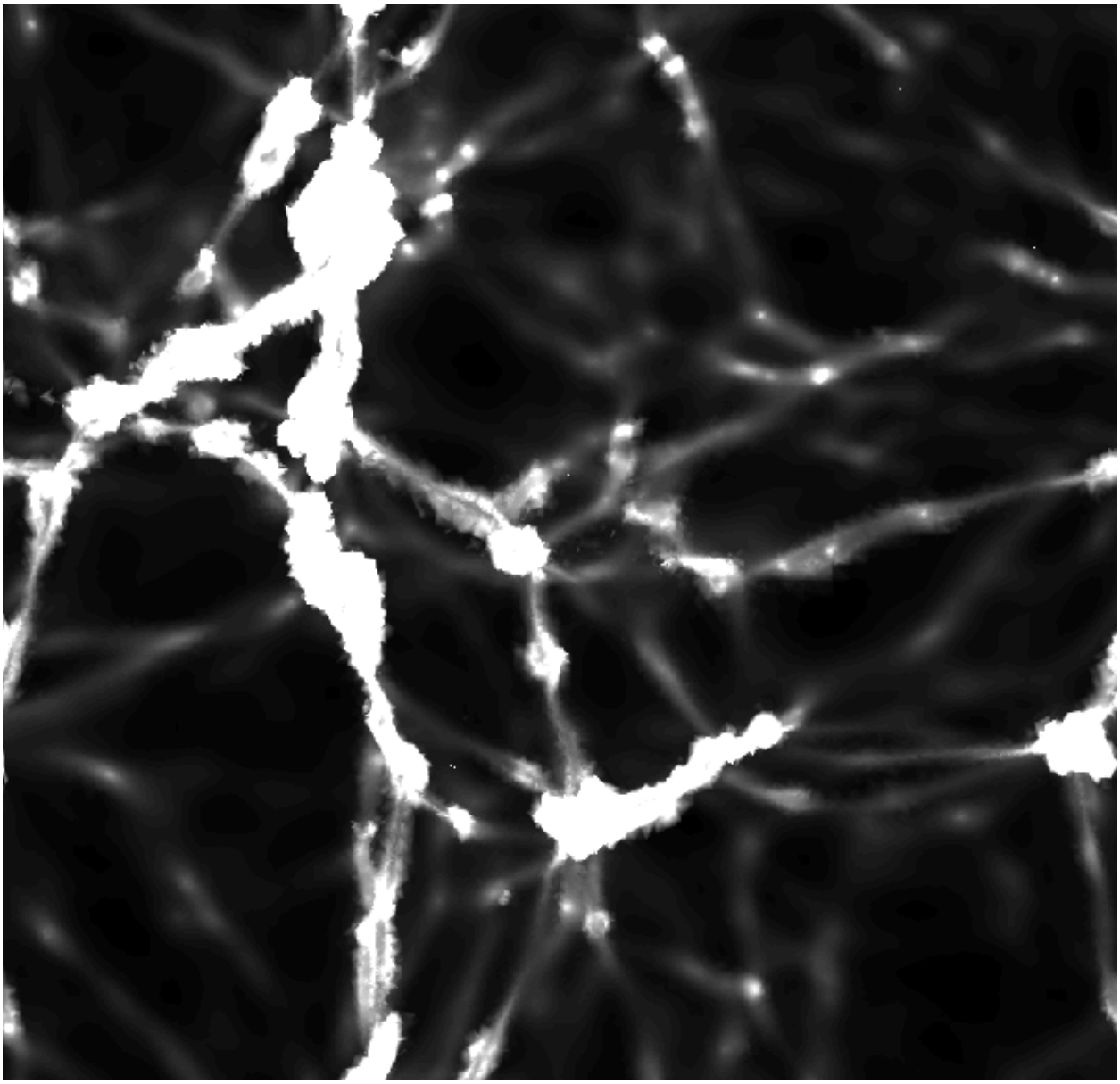

Figure 8.1. Internal energy in the Arepo data at $z = 0.3$ using the DM declustering mechanism

```
// Set up lattice parameters
GBounds geomBounds = new GBounds(lowerGeomBounds, upperGeomBounds);
ISBounds gridBounds = new ISBounds(lowerGridBounds, upperGridBounds);

Approximator approx = new VolumeTetrahedralCellApproximator();
float steps[] = {0.001f,0.001f,0.001f};
int dsorderingarray[] = {0,1,2};
AxisOrdering dsordering = new AxisOrdering(dsorderingarray);
Geometry geometry=new UnstructuredGeometry(gridBounds, geomBounds);
Topology topology=new UnstructuredTopology(geometry, dimensionality);

// Create and activate lattice
Lattice myLattice = new Lattice("Arepo", topology, geometry, approx);

myLattice.activate();

// Generate lattice image
GIterator iter = new GIterator(geomBounds,dsordering,steps);
float dataValues;
for(iter.init(); iter.valid(); iter.next()) {
   dataValues = myLattice.simplexDatum(iter);
   plotPoint(iter, dataValues);
}
```

Figure 8.2. Creating and Visualizing a Lattice

query point. That is, we can view a tetrahedron as a collection of four triangular facets that each define a plane. For each plane, the tetrahedron is located entirely on one side (the *in* side), and not the other. If a query point is found to be on the *in* side for all four facets, then it must be inside the tetrahedron.

For this approach to work, we must be able to reliably identify the *in* side of each facet. Since each facet is defined by three vertices, we currently do this by noting on which side of the facet the fourth vertex lies. In future, we also expect to be able to take advantage of a standard *winding order*, which would reduce computation, further enhancing the effectiveness of our I/O optimizations.

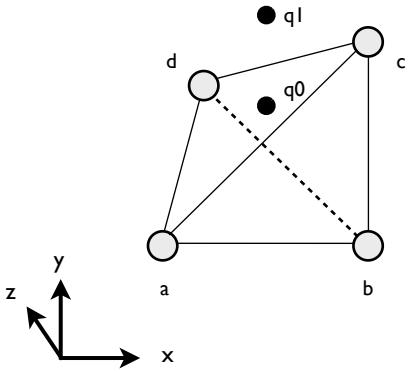

Figure 8.3. A tetrahedron with query point q_0 occurring inside the tetrahedron and query point q_1 occurring outside the tetrahedron

Figure 8.3 illustrates our current approach. It shows a tetrahedron and how the query point is evaluated for a plane. Evaluating the plane equation for facets $\{a,b,c\}$, $\{a,c,d\}$ and $\{a,b,d\}$, both query points q_0 and q_1 will have the same sign, either positive or negative. Query point q_0, which falls inside the tetrahedron, will have the same sign when facet $\{b,c,d\}$ is evaluated. When q_1 is evaluated the sign is different, indicating that the point is not located inside the tetrahedron.

CHAPTER 9

RESULTS

9.1 Disk Level Partitioning Results

We perform our tests on a 64bit Linux machine running Linux version 2.6.18 with Intel Xeon® processors with 16 cores, each running at 2.4GHz. The system has 24GB of memory, but our system used a maximum of only 1.6GB. We perform our tests using a dataset of 15GB for the two dimensional dataset and 25GB for the three dimensional dataset. We generate 2D and 3D unstructured grids using a seeded random number generator to simulate the vertex points. The vertices lies within a domain space of 0.0 to 1.0. The delaunay triangulation is performed with the open source software *qhull* [14]. The 2D dataset contains over 31 million tetrahedra, while there are over 67 million tetrahedra in the 3D dataset.

We clear the filesystem cache after each run to ensure fairness across runs. The results presented are an average of at least three runs.

We perform datum query iterations on partitionings of varying dimensions. We perform three types of query tests. The first query tests are performed on the reorganized files generated from the splitting process. The second query tests are performed on the merged file using the LRU cache mechanism and the third query tests are performed on the merged file using IAP cache. We refer to these tests as *reorganized*, *merged* and *IAP cache* (or simply

cache) respectively in our results. We use several iteration orderings for the 2D and 3D tests. We use orderings {0,1} and {1,0} for the 2D partitioning and {0,1,2} and {2,1,0} for the 3D partitioning.

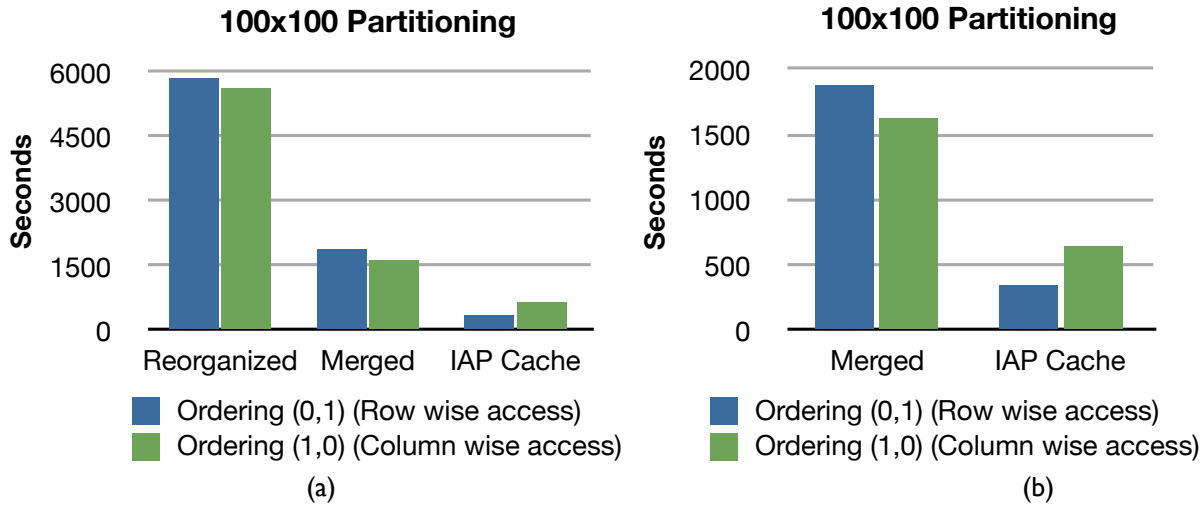

Figure 9.1. Datum Iteration with a step of 0.01 and 100x100 partitioning

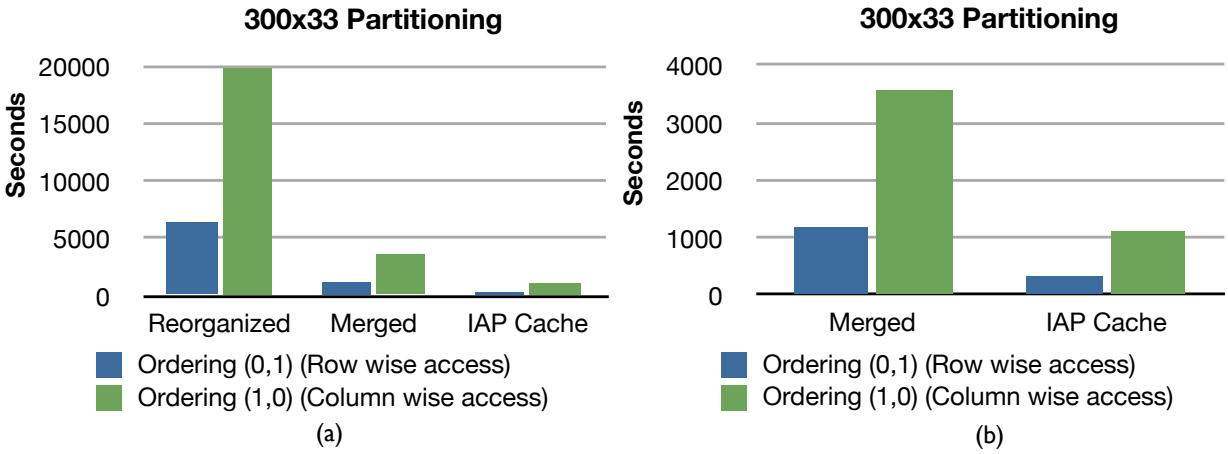

Figure 9.2. Datum Iteration with a step of 0.01 and 300x33 partitioning

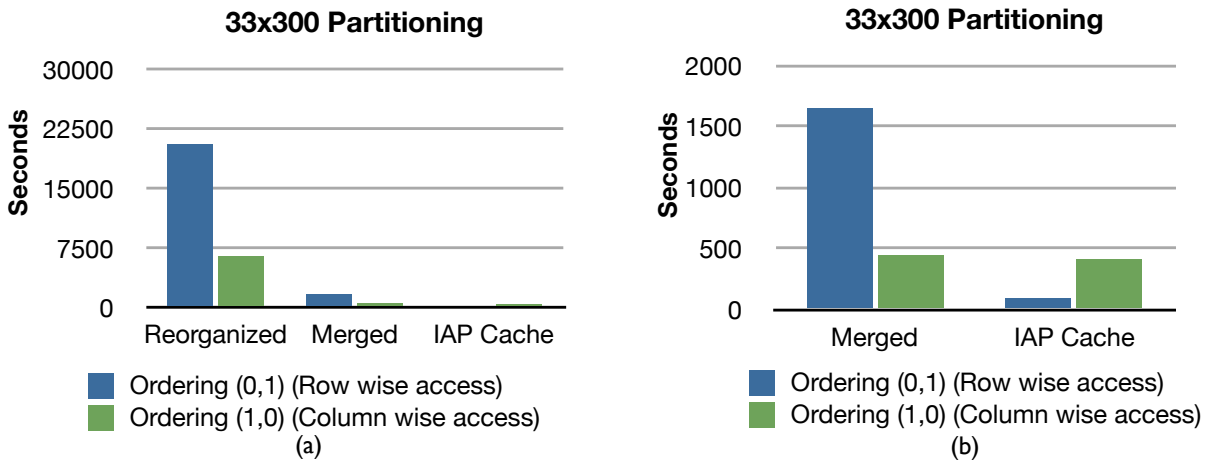

Figure 9.3. Datum Iteration with a step of 0.01 and 33x300 partitioning

9.1.1 Datum Query Iteration over 2D Files

We perform datum queries using a two dimensional file with an equal sided partitioning of 100x100 and unequal sides 300x33 and 33x300. We use an 8x8 subgrid partitioning and a sampling density of 0.01 for the iteration (i.e. we query the domain space at every 0.01 interval). Figures 9.1*a*, 9.2*a* and 9.3*a* show our results for the 100x100, 300x33 and 33x300 partitioning respectively. Figures 9.1*b*, 9.2*b* and 9.3*b* shows only the merged and cache portion of the results. We perform datum queries using the separate partition files generated from the data reorganization, the resulting merged file, and the cache mechanism.

The merged file shows significant gain in performance when compared to the reorganized data files both in the orderings {0,1} and {1,0}. With the row wise access, it takes about 97 minutes to iterate over the reorganized files and about 31 minutes for the merged file using the 100x100 partitioning. The cache performs far better as it takes about 6 minutes for the datum query iteration. When iterating in a column wise fashion, the reorganized data completes in about 94 minutes and the merged file takes approximately 27 minutes. When we utilize the cache, performance is further improved as the iteration takes about 11 minutes to complete resulting in a speedup of 8.73. Table 9.1 shows the speedup results for the 2D dataset when we compare against the reorganized files.

Ordering	Row wise			Column wise		
	100x100	300x33	33x300	100x100	300x33	33x300
Merged	3.12	5.43	12.44	3.45	5.54	14.50
Cache	16.92	19.75	236.7	8.73	17.79	15.76

Table 9.1. Speedup Results for 2D dataset

In both the 300x33 and 33x300 partitioning, the cache performs better than the merged file and reorganized approach. The cache shows tremendous speedup particularly in the 33x300 partitioning when accessing the data in a row wise pattern. This is because the access ordering is tuned to the storage ordering and more cell groups are loaded per cache slice exhibiting tremendous performance gain.

Since each cell group is visited at least once, the merged file performs better than the reorganized files. The increased locality of the merged file allows for reduced disk access and takes advantage of the filesystem prefetching. Overall, the cache speedup for the row wise access performs better than the column access because we access the data in a manner consistent with the storage ordering.

9.1.2 Datum Query Iteration over 3D Files

We also perform datum queries over a cubic 3D file using a cubic partitioning of 10x10x10 and partition configurations 18x10x5, 10x18x5 and 10x10x20. These partitioning configurations produce partition elements that are similar in shape to the diagrams shown in figure 3.10. We perform tests using different subgrid configurations for the dataset partitioning. We use a subgrid of 2x2x2 and a subgrid of 8x8x8 and use a sampling density of 0.01.

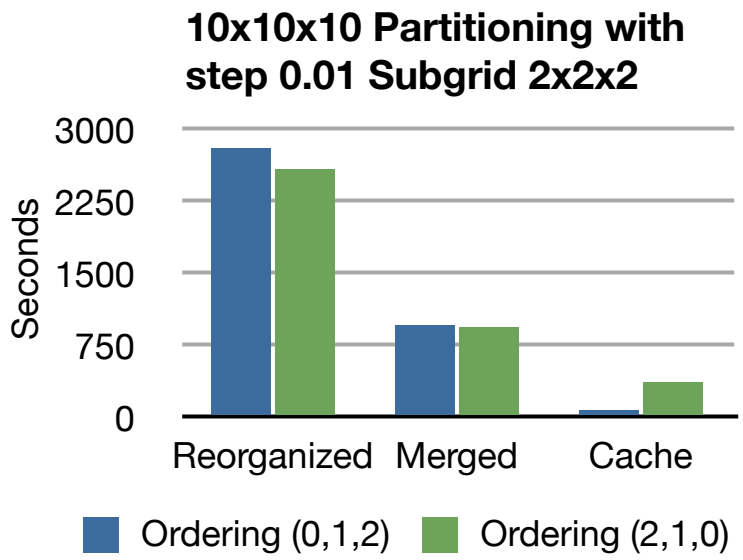

Figure 9.4. Datum Iteration with a step of 0.01, 10x10x10 partitioning and subgrid 2x2x2

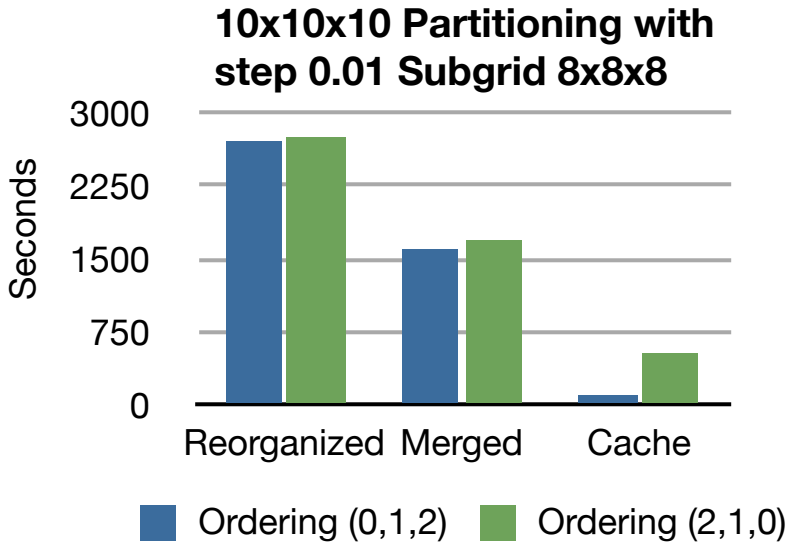

Figure 9.5. Datum Iteration with a step of 0.01, 10x10x10 partitioning and subgrid 8x8x8

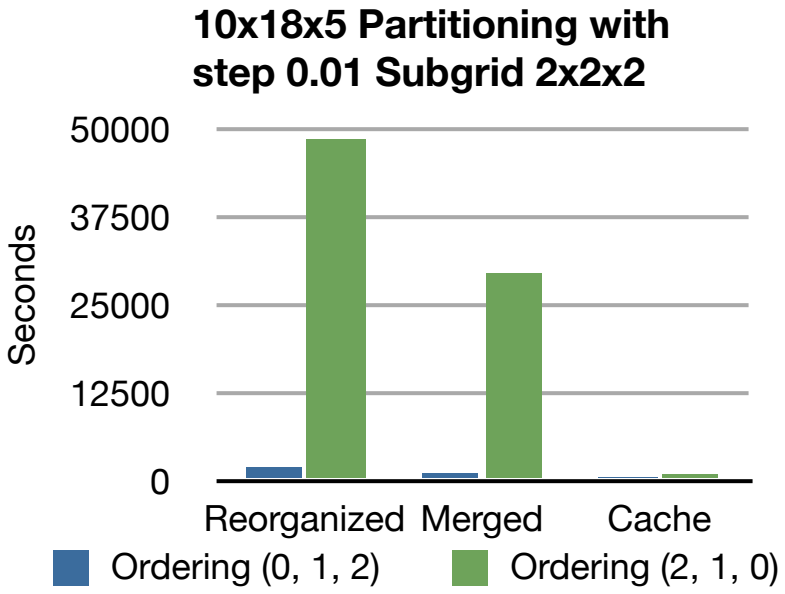

Figure 9.6. Datum Iteration with a step of 0.01, 10x18x5 partitioning and subgrid 2x2x2

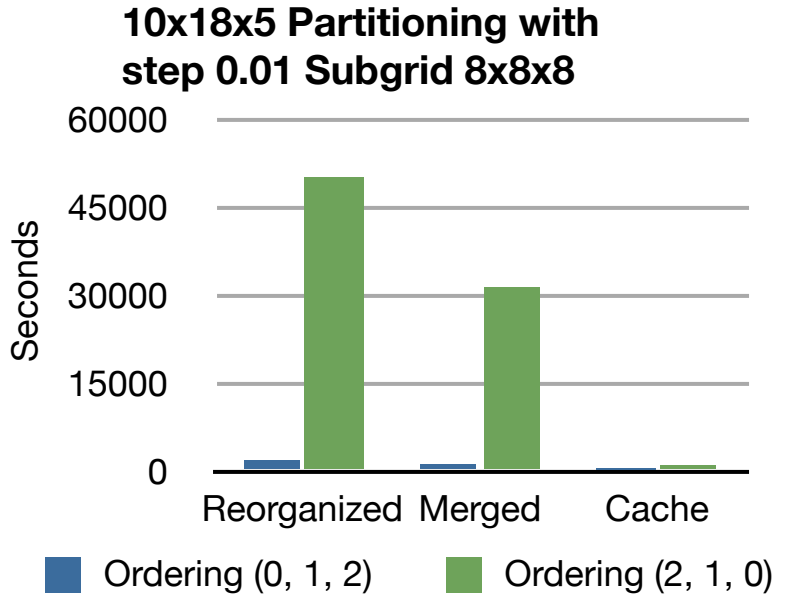

Figure 9.7. Datum Iteration with a step of 0.01, 10x18x5 partitioning and subgrid 8x8x8

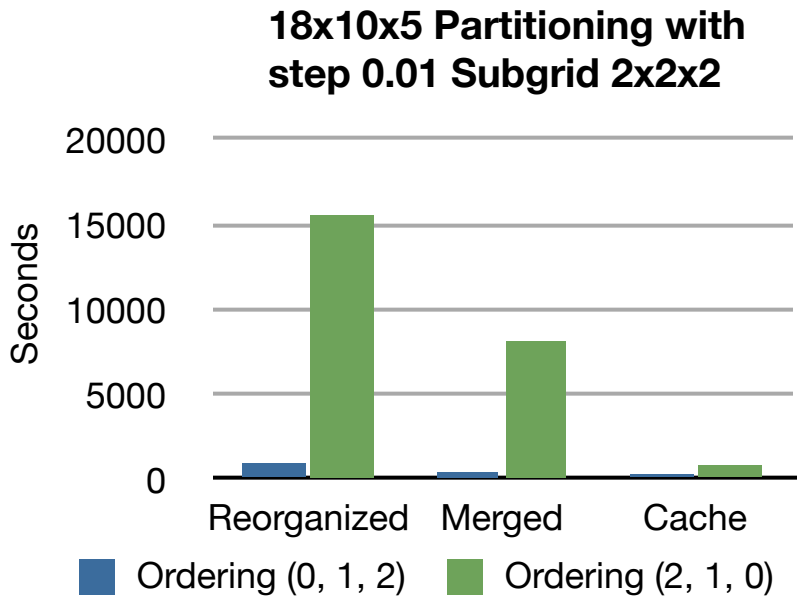

Figure 9.8. Datum Iteration with a step of 0.01, 18x10x5 partitioning and subgrid 2x2x2

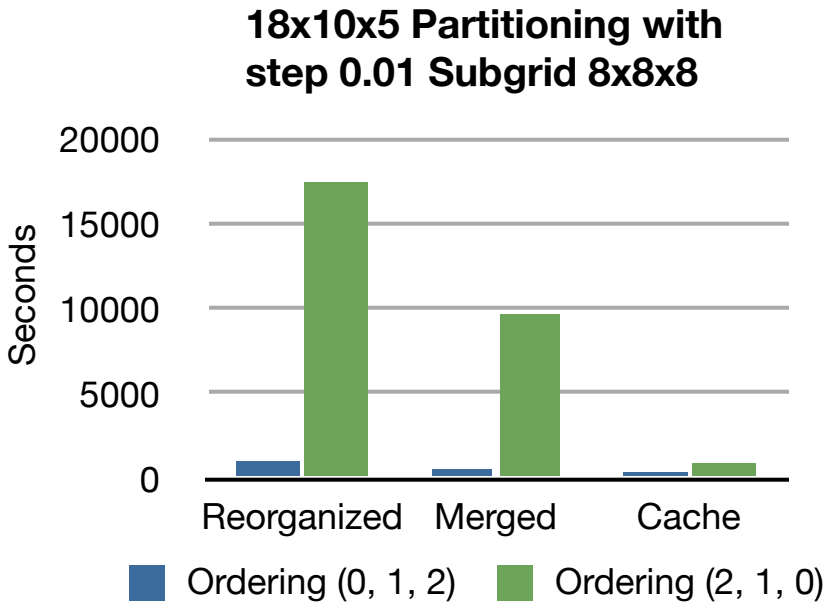

Figure 9.9. Datum Iteration with a step of 0.01, 18x10x5 partitioning and subgrid 8x8x8

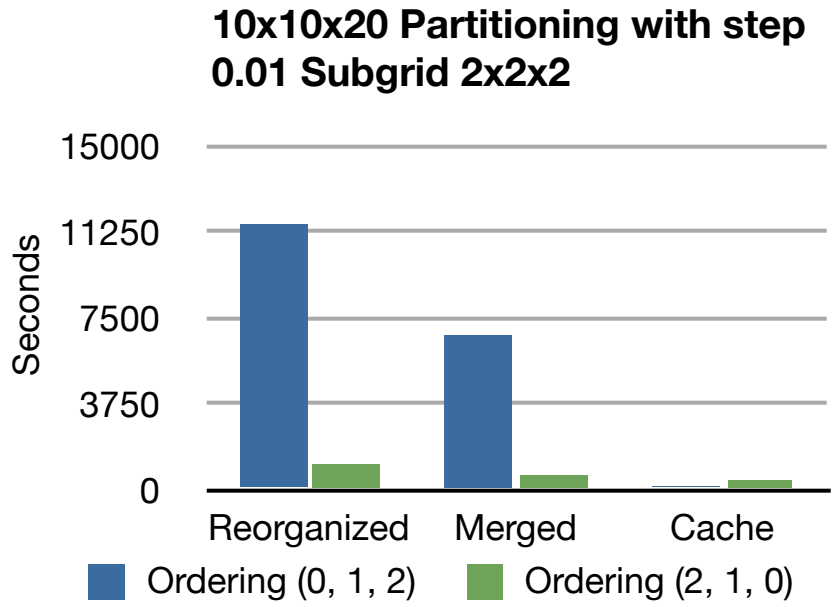

Figure 9.10. Datum Iteration with a step of 0.01, 10x10x20 partitioning and subgrid 2x2x2

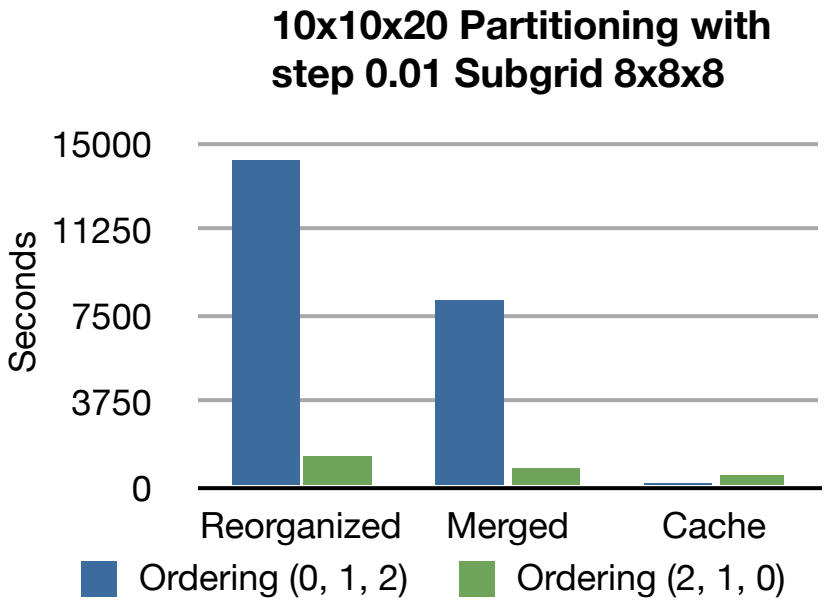

Figure 9.11. Datum Iteration with a step of 0.01, 10x10x20 partitioning and subgrid 8x8x8

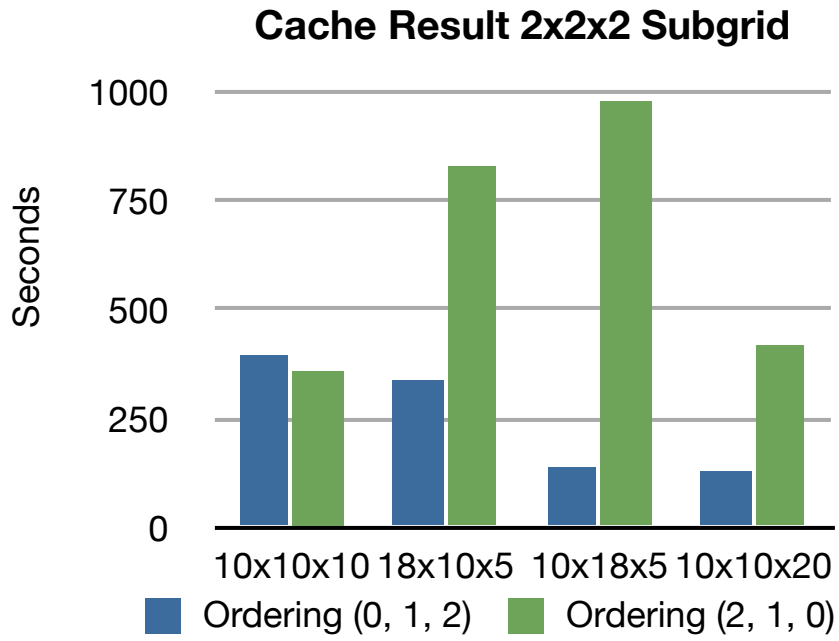

Figure 9.12. Cache results for partitioning 10x10x10, 10x18x5, 18x10x5 and 10x10x20 subgrid 2x2x2

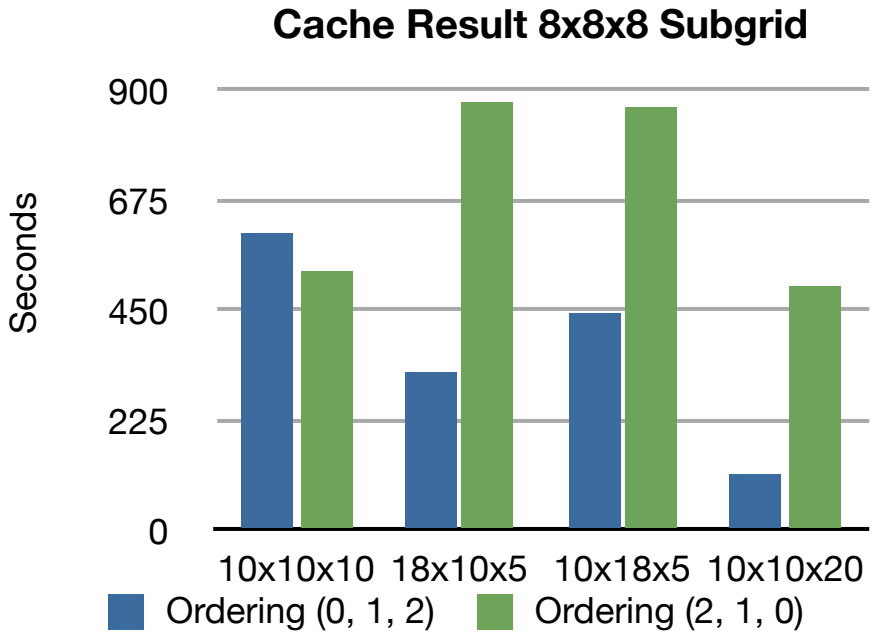

Figure 9.13. Cache results for partitioning 10x10x10, 10x18x5, 18x10x5 and 10x10x20 subgrid 8x8x8

Ordering	Row wise		Column wise	
	2x2x2 Subgrid	8x8x8 Subgrid	2x2x2 Subgrid	8x8x8 Subgrid
Merged	2.94	1.69	2.76	1.62
Cache	42.44	27.38	7.21	5.19

Table 9.2. Speedup Results for 10x10x10 3D dataset

Ordering	Row wise		Column wise	
	2x2x2 Subgrid	8x8x8 Subgrid	2x2x2 Subgrid	8x8x8 Subgrid
Merged	1.73	1.69	1.65	1.60
Cache	5.10	5.65	58.46	57.21

Table 9.3. Speedup Results for 10x18x5 3D dataset

Figure 9.4 shows the results for a cubic partitioning of 10x10x10 using a 2x2x2 subgrid and figure 9.5 shows the results for a cubic partitioning of 10x10x10 using an 8x8x8 subgrid. The cache and merged file performance are compared to the reorganized file and the speedup values are shown in table 9.2. Figures 9.6 and 9.7 shows the results for the partitioning 10x18x5 using subgrids 2x2x2 and 8x8x8 respectively. Figure 9.8 shows the results for the partitioning 18x10x5 using subgrid 2x2x2 and figure 9.9 shows similar result using subgrid 8x8x8. Figures 9.10 and 9.11 shows the results for the partitioning 10x10x20 using subgrids 2x2x2 and 8x8x8 respectively.

We experience a greater speedup value for the cache when we utilize a row wise access pattern. The 2x2x2 subgrid however tends to perform better than the 8x8x8 subgrid in both row wise and column wise access patterns. This is because the granularity significantly decreases in the 8x8x8 subgrid. This introduces some overhead evident in the result when we perform a datum query. However, the coarse 2x2x2 subgrid performs better as fewer cells span partition boundaries and the overhead is reduced, therefore showing significant performance gain. Figures 9.12 and 9.13 compares the cache results of the different 3D partitioning. The result shows an overall better performance when we utilize a row wise

Ordering	Row wise		Column wise	
	2x2x2 Subgrid	8x8x8 Subgrid	2x2x2 Subgrid	8x8x8 Subgrid
Merged	1.62	2.60	1.62	2.46
cache	20.98	16.53	27.92	77.94

Table 9.4. Speedup Results for 18x10x5 3D dataset

Ordering	Row wise		Column wise	
	2x2x2 Subgrid	8x8x8 Subgrid	2x2x2 Subgrid	8x8x8 Subgrid
Merged	2.53	2.52	2.68	2.55
cache	515.01	595.25	81.52	69.96

Table 9.5. Speedup Results for 10x10x20 3D dataset

access pattern over the column wise access pattern as we access the data following the ordering the data is stored with.

Overall, the cache outperforms the merged file format and the reorganized files in both the two dimensional and three dimensional files. Also, the merged file performs better than the reorganized files in all cases. When using either a row wise or column wise access pattern, the cache shows significant gain in performance compared to the merged file and reorganized files. Accessing the data in the manner it is stored results in much greater performance.

9.1.3 Datum Query Iteration over Real World Data

We apply our approach and perform datum query iteration using real world scientific data. The data is generated from fluid simulations where space is decomposed based on a Delaunay tessellation. The mesh generating points are from the *Arepo* series of simulations described in Nelson et al [46]. The dataset contains over 11 million particles that represent fluid elements and have associated fluid density, energy and velocity (x,y,z) data values.

Converting this dataset to our merged format, we generate the tetrahedral mesh,

containing over 75 million tetrahedra. Since these tetrahedral cells are essential for most scientific applications, the processing and storage requirements of this mesh are largely unavoidable. However, some further storage is required by our owner–borrower scheme. For this data set, our scheme increases the storage requirements by 16.5%. However, even this modest percentage will be further reduced when the number of data attributes per vertex is increased. For example, assuming the data attribute portion increases by a factor of 10, the owner–borrower information added will remain the same and the extra storage requirement percentage required falls to 2.3%.

Figure 9.14 and 9.15 shows our results using subgrids 2x2x2 and 8x8x8 respectively. Table 9.6 shows the speedup achieved. The cache shows tremendous performance gains compared to the reorganized files and the merged file.

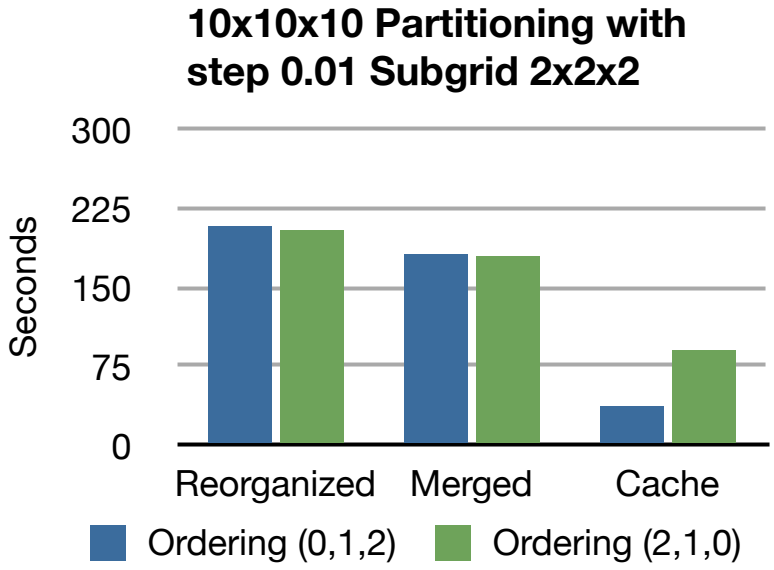

Figure 9.14. Datum Iteration with a step of 0.01, 10x10x10 partitioning and subgrid 2x2x2 using real data

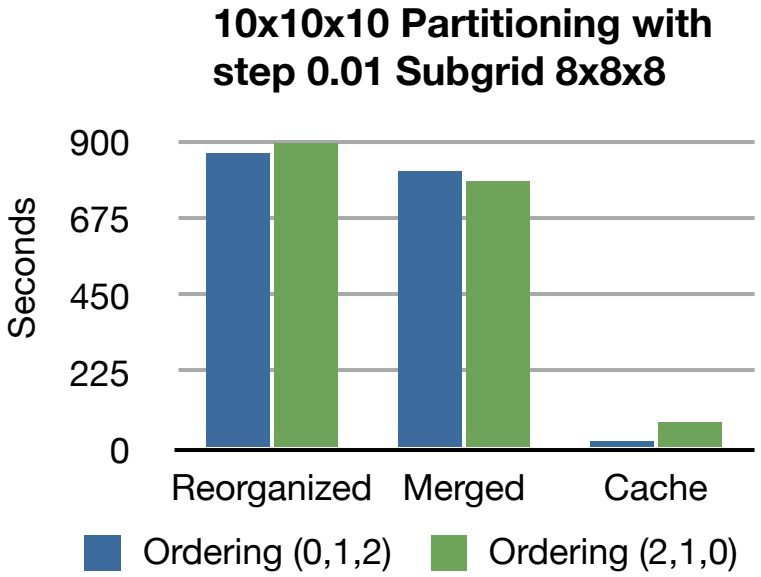

Figure 9.15. Datum Iteration with a step of 0.01, 10x10x10 partitioning and subgrid 8x8x8 using real data

Ordering	Row wise		Column wise	
	2x2x2 Subgrid	8x8x8 Subgrid	2x2x2 Subgrid	8x8x8 Subgrid
Merged	1.15	1.07	1.13	1.14
cache	5.69	39.24	2.26	11.34

Table 9.6. Speedup Results for real world dataset

9.1.4 Discussion

The results section does not compare performance between datasets before and after reorganization because access times required before reorganization were impractically large. Even without the merging step, the improvement in locality provided by partitioning the dataset into cell groups brought traversal times into feasible range, from more than half a day down to an hour and a half.

In addition to this fundamental improvement, the partitioning allows us to construct a slice of the iteration space that is aligned with any of the major axes, and can be read efficiently in a small number of read transactions from disk. It is this facility that allows the

creation of an IAP cache, which reconciles the user access pattern with the manner in which data is stored on disk.

9.2 In Memory Partitioning Results

We perform tests that vary the in-memory partitioning and analyzed the effects on performance. When examining the in-memory partitioning, we analyze performance when we access data using LRU cache and IAP access methods on the merged file and present our results in this section. The merged file provides improved locality within the one dimensional file compared to the reorganized file generated from the disk level partitioning. When evaluating the declustering approach, we must use reorganized files and we compare our results when we access data using reorganized files and when we use parallel IAP. We are able to significantly reduce latency costs by paying the cost once while prefetching multiple partition elements in parallel from multiple disks.

The tests are performed on a 64bit Linux machine running Linux version 2.6.18 with Intel Xeon® processors with 16 cores, each running at 2.4GHz. The system has 24GB of memory, but our software used a maximum of only 1.6GB. We perform tests using a synthetic 3D dataset produced using a seeded random number generator to generate the vertex points. The delaunay triangulation is performed with the open source software *qhull* [14]. The vertices lies within a domain space of 0.0 to 1.0. The dataset is 25GB and contains over 67 million tetrahedra.

The filesystem cache is cleared after each run to ensure fairness across runs. The results presented are an average of at least three runs.

9.2.1 In-Memory Results

We performed in-memory partitioning using different configurations starting with a coarser partitioning of 2x2x2 and progressively finer grained partitioning up till 11x11x11. We use

a 10x10x10 disk level partitioning. The results are as shown in figures 9.16 and 9.17.

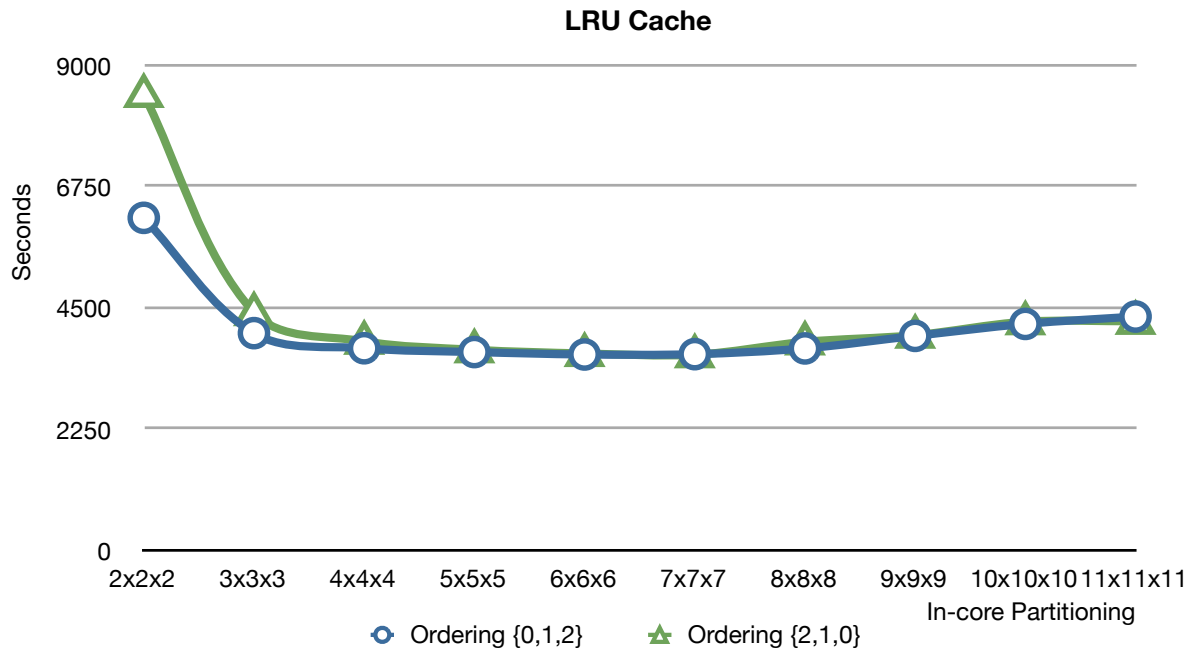

Figure 9.16. Effects of varying in-memory partitioning using LRU with the merged file containing synthetic data

Analyzing the results, performance suffers with coarser partitioning 2x2x2 and 3x3x3. This is as a result of the large number of candidate cells contained in each partition. As the partitioning becomes finer grained, we see a much improved performance in both LRU cache and IAP cases. Performance begins to deteriorate in the LRU cache as the in-memory partitioning approaches 9x9x9, 10x10x10 and 11x11x11. This is because of the increased boundary cell cases, as more cells span partition boundaries. The increased boundary cell cases causes the overall size of the merged file to increase about 0.4% as the in-memory partitioning increases in granularity. On average, performance is best when the partitioning configurations 5x5x5 and 6x6x6 are used.

There is a dramatic initial improvement in performance with IAP as the in-memory partitioning becomes finer grained. However, further increasing the fineness yields diminishing gains in performance and only serves to increase the size of the overall dataset due

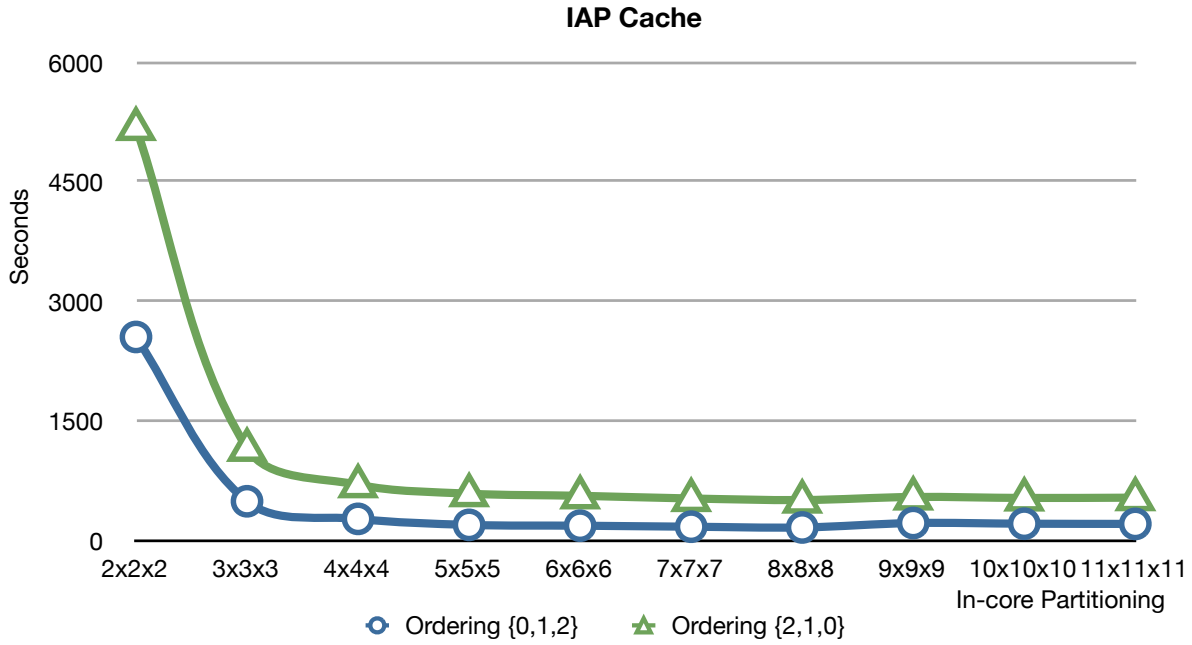

Figure 9.17. Effects of varying in-memory partitioning using IAP with the merged file containing synthetic data

to the increase in borrowed cells across partition boundaries. IAP is able to maintain good performance because of the prefetching mechanism and the fact that IAP amortizes the latency cost of reading owner partitions from disk into memory. However, in the LRU cache case there is more individual loading of owner partitions.

9.2.2 In-Memory Real Data Results

We test our in-memory partitioning using real data. The dataset contains over 11 million particles that represent fluid elements and have associated fluid density, energy and velocity (x,y,z) data values and has over 75 million tetrahedra. The results are presented in figures 9.18 and 9.19. Similar results are achieved using the real data as performance is best around the 5x5x5 and 6x6x6 in-memory partitioning.

In determining an efficient coarseness, there is a tradeoff between the size of the partition element and the number of cells that span partition boundaries. From our results,

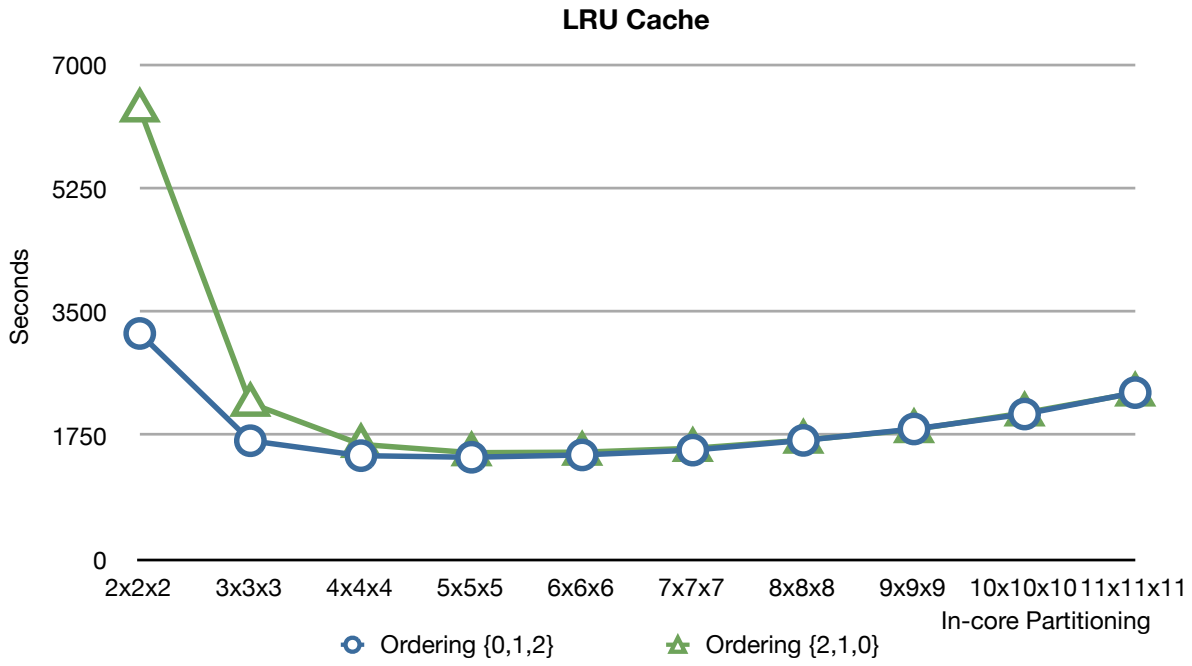

Figure 9.18. Effects of varying in-memory partitioning using LRU with the merged real data file

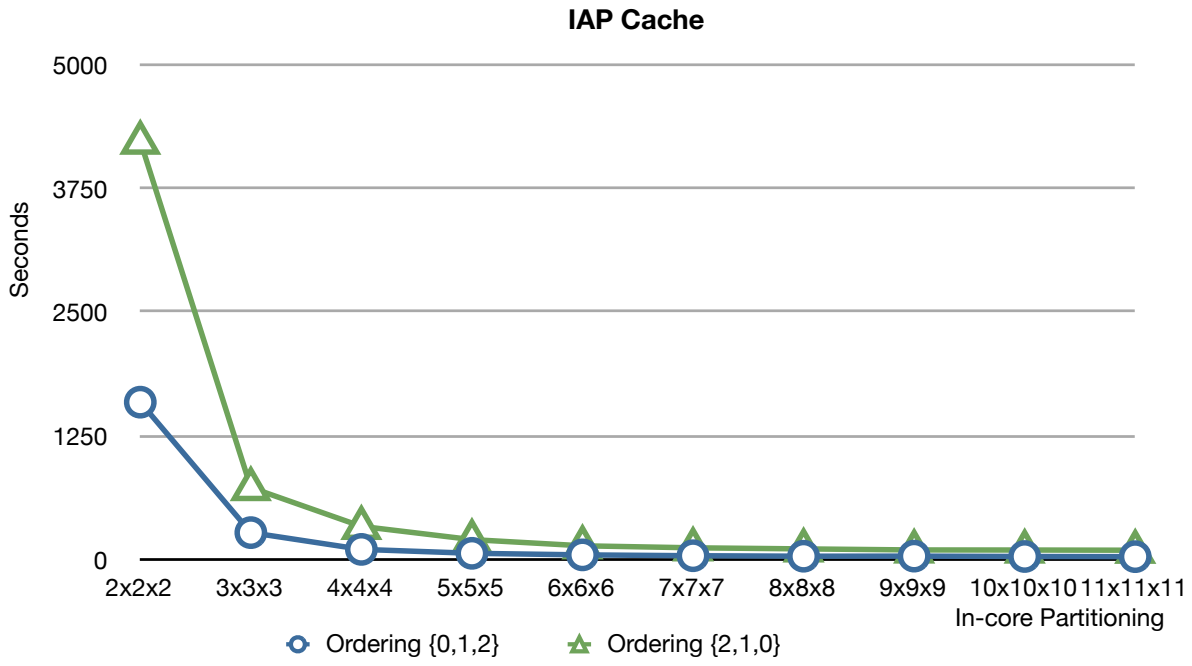

Figure 9.19. Effects of varying in-memory partitioning by applying IAP to real data

an efficient number of cells per in-memory partition is around $n = 300...400$ cells when using LRU caching. This is the point where we find the best tradeoff between all of the competing factors, such as the granularity of the partitioning, the cost of dereferencing borrowed cells, and loading required owner partitions. Dereferencing the borrowed cells involves loading the owner partition from disk in order to access the data values associated with the borrowed cell from the owner partition.

The same value for n performs well for IAP, but we can get additional gains by choosing a finer grid if we know that IAP will be used. In either case, it is easy to experimentally determine n. In the distributed case, we envision being able to tune n to the storage hardware.

9.3 Varying In-Memory partitioning configuration

Based on our results from the previous section that gives us an idea of the efficient number for n which is the number cells per in-memory partitioning. We proceed to verify equation 3 and vary the in-memory partitioning configuration from a cubic partitioning to the different shapes shown in figure 3.10 while maintaining $n = 300...400$. For clarity, we refer to the shapes in figures 3.10*b*, 3.10*c* and 3.10*d* as *wide*, *tall* and *deep* respectively.

The results shown in figures 9.20, 9.21 and 9.22 are obtained using a disk level partitioning 10x10x10. The data is accessed using the LRU cache described in section 6.2. The in-memory partitioning configurations used in figure 9.20 generates a wide shaped in-memory partition element. We derived the ideal in-memory partitioning for this case using equation 5.3 while maintaining the wide shaped in-memory partitioning. This results in a partitioning configuration 12x6x3. We vary this partitioning configuration by making it more coarse and fine grained and the best results are observed when partitioning configuration 12x6x3 is used in both ordering {0, 1, 2} and {2, 1, 0}.

Similarly, using equation 5.3, we derive in-memory partitioning configurations 6x12x3

and 5x5x9 for the tall and wide shaped in-memory partition elements and the results are as shown in figures 9.21 and 9.22 respectively. The partitioning configurations are varied to create coarser and finer grained partition elements of the same shape. The best results are observed with the partitioning configurations derived using equation 5.3 using 6x12x3 for the tall shaped and around 5x5x9 for the deep shaped in-memory partitioning configurations.

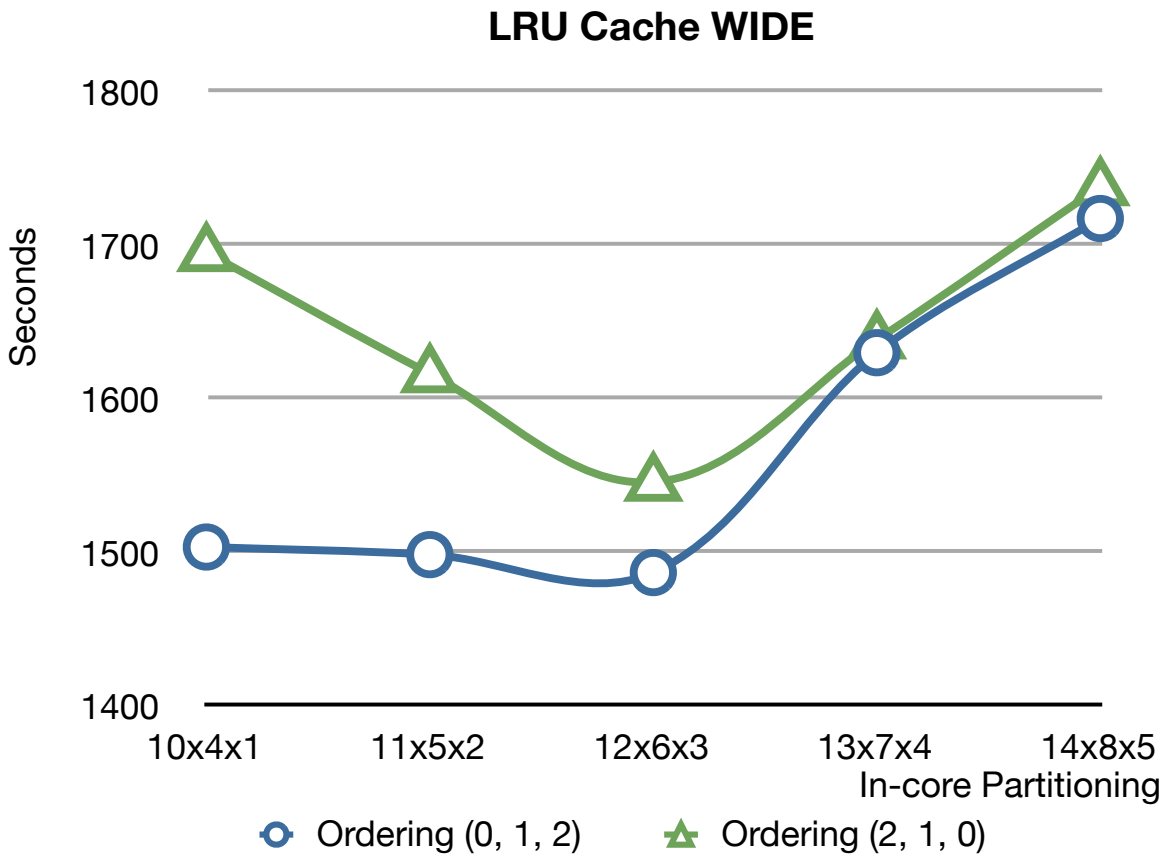

Figure 9.20. Varying in-memory partitioning that creates a wide shaped in-memory partition element. $n = 300...400$ when in-memory partitioning is 12x6x3

When use IAP using the same configurations, our results are shown in figures 9.23, 9.24 and 9.25.

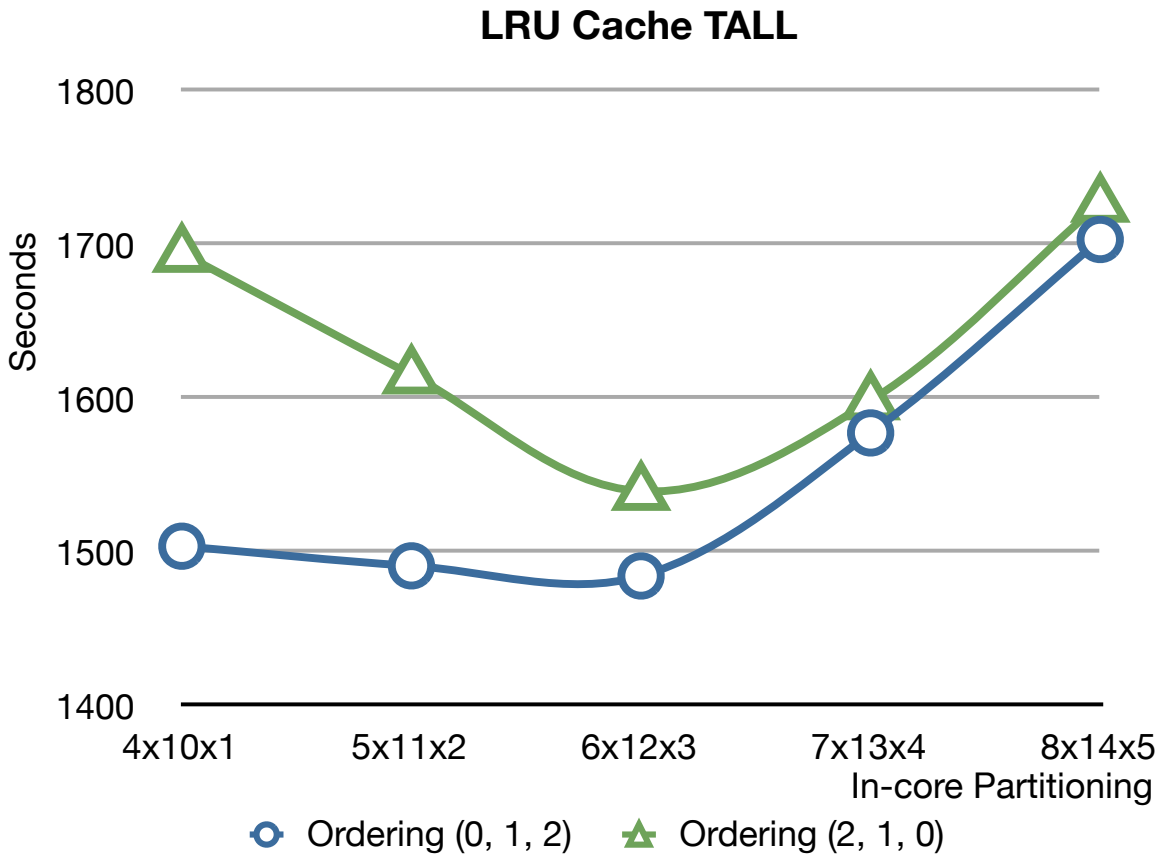

Figure 9.21. Varying in-memory partitioning that creates a tall shaped in-memory partition element. $n = 300...400$ when in-memory partitioning is 6x12x3

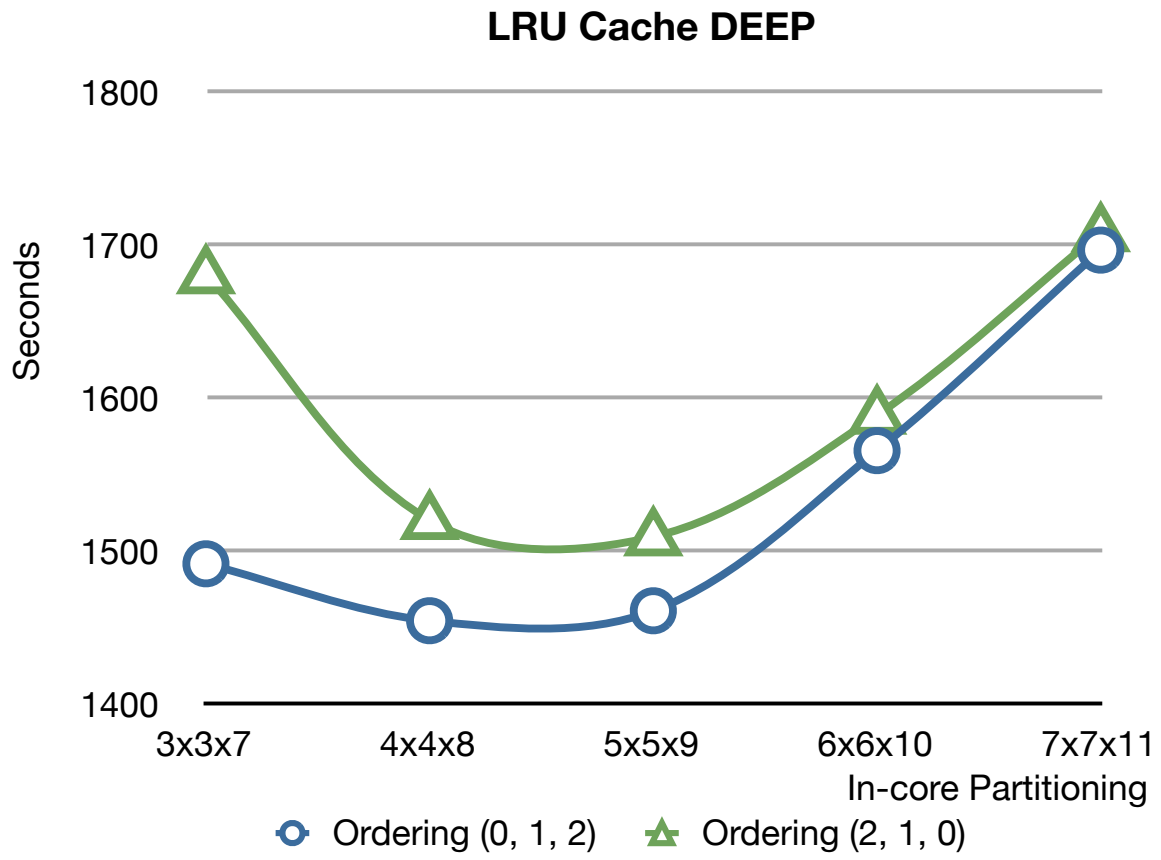

Figure 9.22. Varying in-memory partitioning that creates a deep shaped in-memory partition element. $n = 300...400$ when in-memory partitioning is 5x5x9

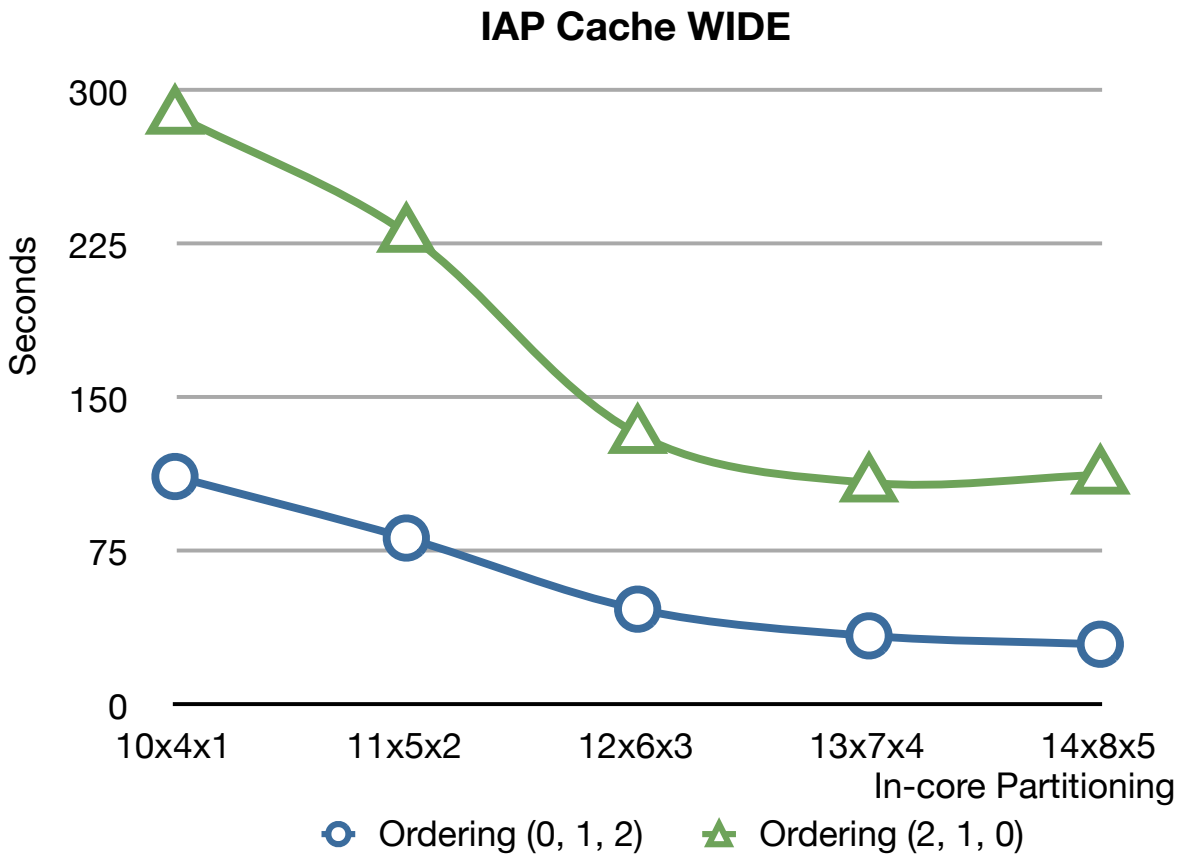

Figure 9.23. Varying in-memory partitioning that creates a wide shaped in-memory partition element and applying IAP. $n = 300...400$

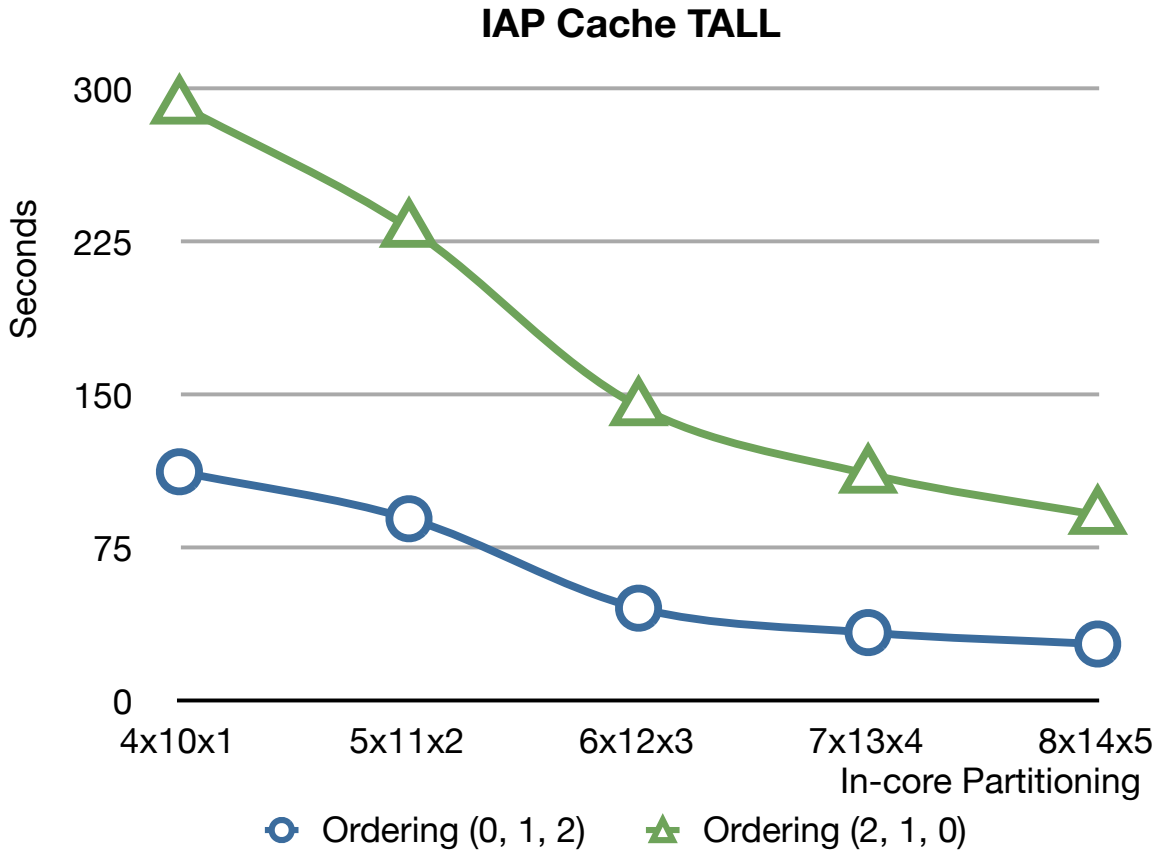

Figure 9.24. Varying in-memory partitioning that creates a tall shaped in-memory partition element and applying IAP. $n = 300...400$

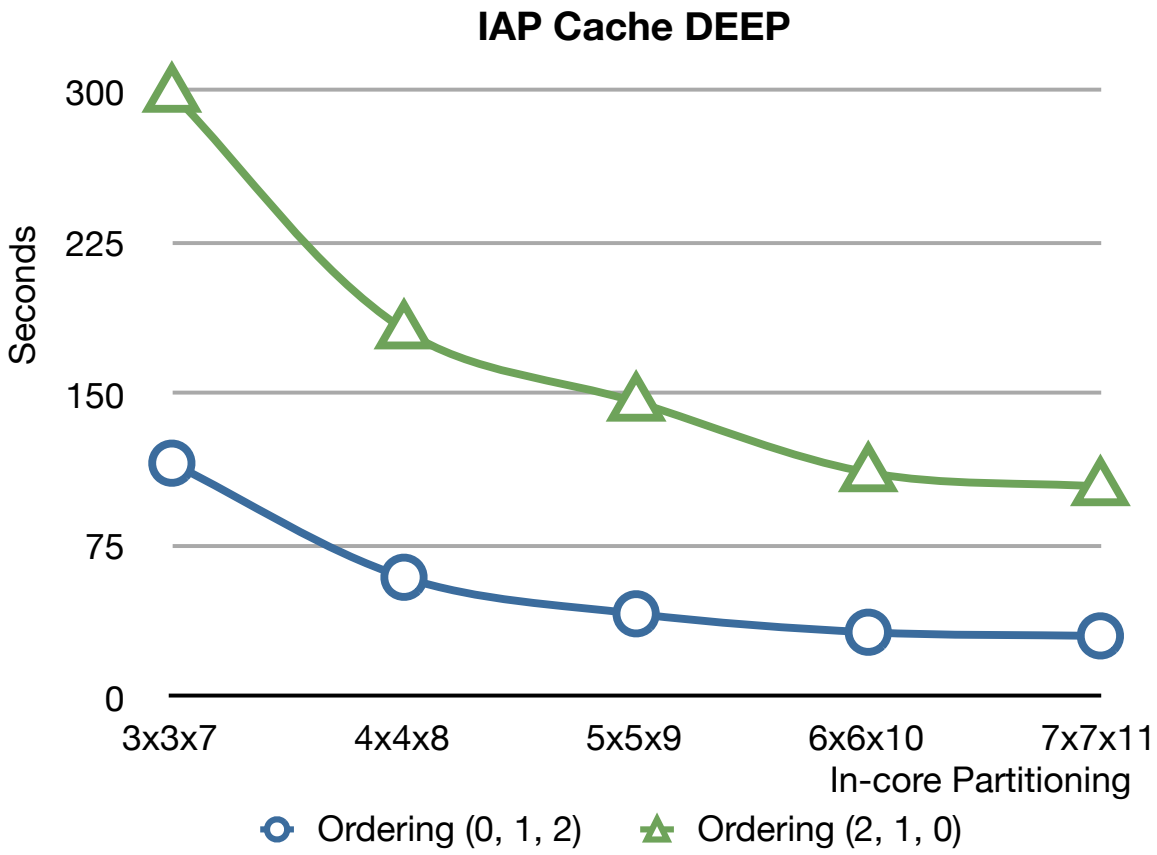

Figure 9.25. Varying in-memory partitioning that creates a deep shaped in-memory partition element and applying IAP. $n = 300...400$

9.4 Declustered Results

We perform similar tests and apply the DM declustering mechanism to access partition elements in parallel using IAP. We use a 10x10x10 partitioning configuration for the disk level partitioning and map the partition elements to 5 disks. From the insights gained by the previous in-memory partitioning tests, we vary the in-memory partitioning used for the declustering mechanism using a larger interval between tests. The results when we apply the declustering mechanism using ordering {0,1,2} and ordering {2,1,0} is shown in figure 9.26. The shape of the graph demonstrates that there is a sweet spot based on the tradeoffs of using both extremes of the partitioning.

We compare our declustering results which prefetches partition elements into an IAP cache with the results of accessing separate partitioned files on a single disk. Our results using a row wise and column wise access pattern are presented in figures 9.27 and 9.28 respectively. Table 9.7 shows the speedup values derived from using the declustered mechanism.

Declustered Speedup				
Subgrid ISBounds	2x2x2	5x5x5	8x8x8	11x11x11
Ordering {0, 1, 2}	2.78	2.07	2.18	3.58
Ordering {2, 1, 0}	3.51	2.06	2.13	4.97

Table 9.7. Speedup Results using declustered IAP

The declustering results show a speed up of 2.06 in the worst case and 4.97 speed up in the best case, which is nearly equal to the number of disks we access in parallel. Overall, we significantly improve the access time of unstructured grids residing on multiple disks while varying the granularity of the in-memory partitioning.

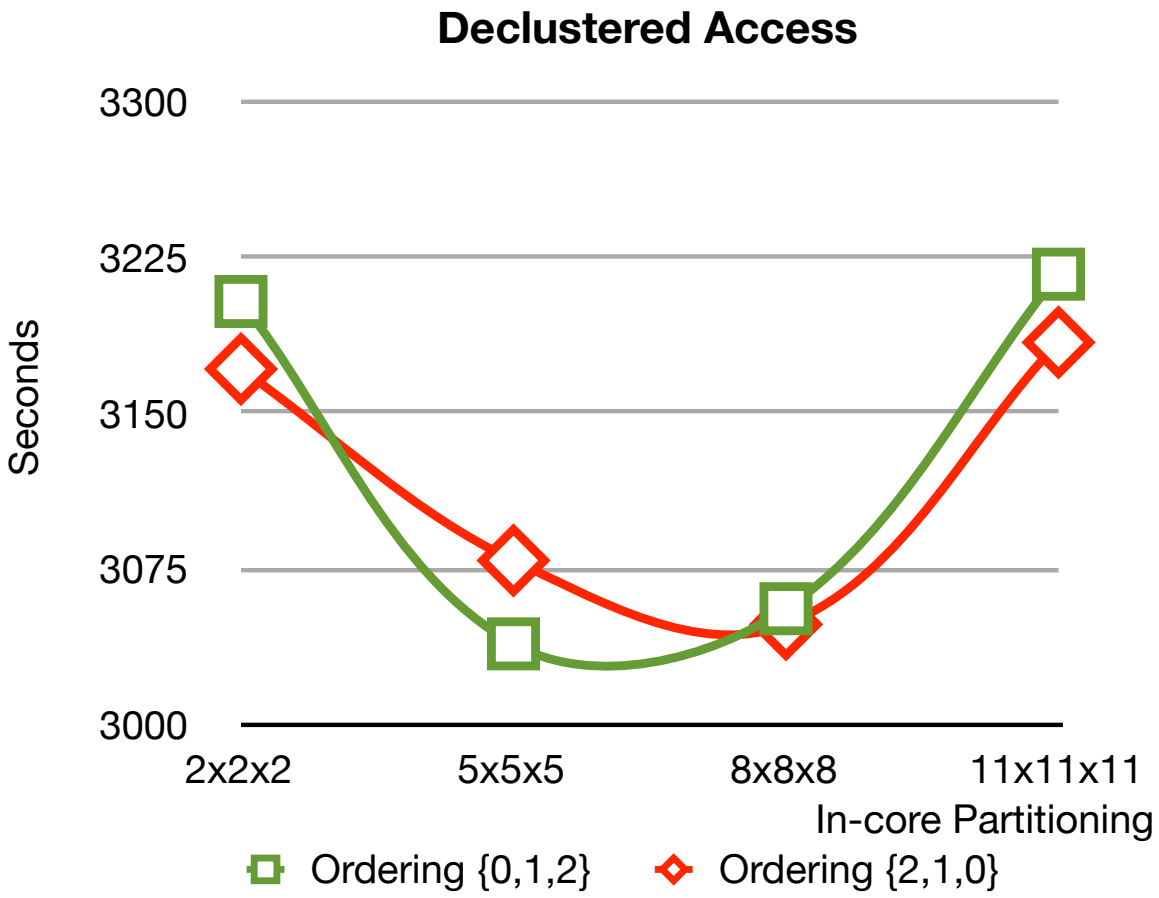

Figure 9.26. Declustered Result using 10x10x10 disk level partitioning and varying the in-memory partitioning

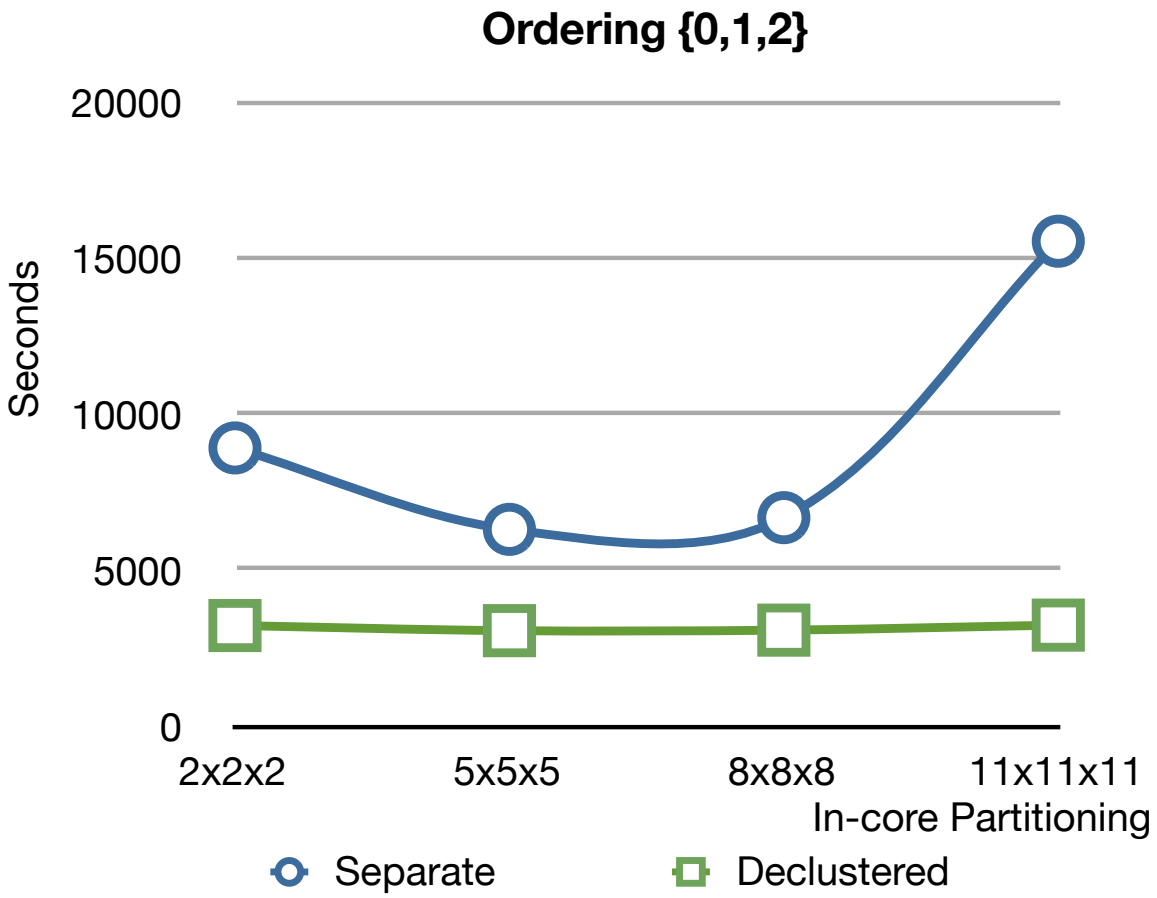

Figure 9.27. Reorganized Files compared with Declustered IAP Result using row wise access pattern (Ordering {0,1,2}) and varying the in-memory partitioning while disk level partitioning is kept constant

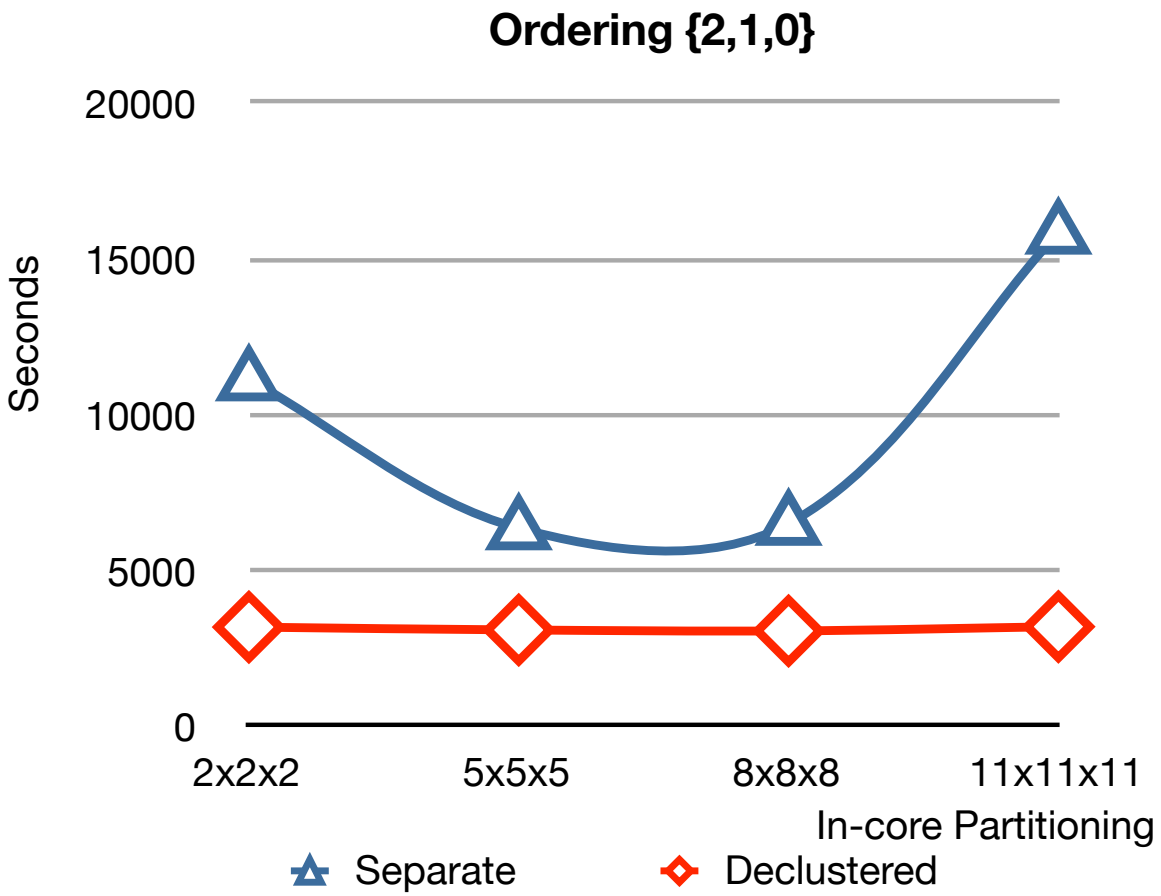

Figure 9.28. Reorganized Files compared with Declustered IAP Result using column wise access pattern (Ordering {2, 1, 0}) and varying the in-memory partitioning while disk level partitioning is kept constant

CHAPTER 10

RELATED WORK

Our work involves two main phases. The first phase involves reorganizing the unstructured grid while maintaining the neighborhood relationship that exists between the vertices of the dataset and improving the file representation on disk. The second phase entails fast retrieval of data from disk using advanced knowledge of the user's access pattern. The rest of this section describes the related work in these areas.

A lot of research work has gone into handling large data and solving issues relating to storage and retrieval of spatial data. However, much of the existing literature have been focused on large multidimensional array data [60, 66]. Data storage and reorganization to facilitate faster retrieval is an important factor when dealing with unstructured grids. As the data grows bigger than memory, the partitioning of the data while maintaining the neighborhood relationship between sample points of the dataset is desirable. Past work by Ueng et al [72], Lindstrom and Pascucci [36], and Childs et al [20] have developed out-of-core approaches for visualization of large datasets. However, some of these approaches are limited due to their on–demand approach to data retrieval. In recent work by Kumar et al [32], they develop a format for fast access to multi-dimensional scientific datasets and parallelizing it to further improve performance. They perform some data reorganization to assist with analysis and visualization. Other work by Ross and Sitaridi [58] propose a method

for accessing multidimensional data with a partitioned *blockmap* index, using a bitmap to achieve minimal space overhead. This work tends to map a bit to a grid which can then be loaded into memory for mapping purposes. However, this approach assumes the data is present in memory and bandwidth is not an issue.

10.1 Spatial Partitioning for Regular Data

Data partitioning is a popular technique used to improve performance when working with very large data sets that will not fit in a single machines memory. Some techniques used in partitioning unstructured grids have stemmed from some of the ideas used to partition large regular datasets. Sarawagi and Stonebraker [63] presents methods for configuring *chunking*, in which data which is nearby in the domain is stored together in chunks on disk. The work by Rotem and Otoo [59, 47] presents and analyzes models for chunking of large multidimensional arrays and provide exact solutions for configuring chunking methods presented by Sarawagi and Stonebraker [63]. Lofstead et al. [39] discussed several reading patterns for large scale I/O. They describe a log based storage approach that organizes data into chunks called *data districts* that facilitates parallel reading and writing of data to and from storage respectively. These past work are however focused on regular data. Tian et al. [67] propose a two-level data reorganization approach similar to what we do, called Smart-IO that improves the retrieval of multidimensional scientific data. The first level divides the data into data chunks of different sizes, while the second level involves data reordering and placement based on a space filling curve. Smart-IO should be applicable to unstructured grids if the chunking it depends upon is extended to handle the special challenges of unstructured grids. Indeed, our own work attempts to make methods first developed for regular data applicable to unstructured grids by addressing these special challenges at the partitioning level.

The disadvantage of such methods that handle multidimensional regular data partitioning is that they may not be easily applied to unstructured grids because of cells that

span partition boundaries and the neighborhood effect of unstructured grids in the data space and in the underlying one dimensional file space.

Partitioning large multidimensional arrays has been extensively studied [63, 59, 47, 60, 66]. A large part of the existing work have handled approaches that can be categorized under disk level partitioning. Recent work by Ross and Sitaridi [58] propose a method for accessing multidimensional data with a partitioned *blockmap* index, using a bitmap to achieve minimal space overhead. Lofstead et al. [39] discussed several reading patterns for large scale I/O. They describe a log based storage approach that organizes data into chunks called *data districts* that facilitates parallel reading and writing of data to and from storage respectively.

Our past work [9] focused on disk level partitioning and providing efficient storage and retrieval mechanism for unstructured grids. We investigated prefetching of unstructured grid from a single disk perspective. In memory partitioning of multidimensional data [77, 11] has been researched but most of the work has been done at the hardware level to improve performance. Cignoni et al. [21] present an out-of-core approach for mesh simplification. In order to improve memory representation, approaches such as iterative improvement [80] and other forms of multilevel partitioning [62, 33, 44] are used.

10.1.1 Data Storage

The manner in which data is stored on disk is an important factor when dealing with spatial datasets. The column–oriented approach for data storage has been a topic of significant research. Tools such as Fastbit [79] and other database management systems such as Vertica [6], Amazon Redshift [1] and Sybase IQ [42] all use this approach. The work by Harizopoulos et al [28] proposes a column based approach for data storage as opposed to a row storage. By prefetching the data, they recorded better disk bandwidth performance, and columns that are not needed can be skipped, thereby improving bandwidth. This approach is used for relatively small grid files and will require some modifications in order to be applied to large

unstructured grids. Alagiannis et al [12] present an adaptive storage and access pattern that responds to a query. Although this approach can be leveraged for extremely large datasets, it has not been applied to unstructured grids. The work by Papadomanolakis et al. [49] implement a reorganization method that reorders the simplices just like we do, but they use a space filling curve to reorder the simplices in the data file and uses a modified breadth first search to locate desired simplices. The space filling curve improves locality of access very significantly, but does not directly provide a mechanism for reading subsets of the data file. Such functionality is more easily provided by reorganization methods that partition simplices into chunks that are stored together on disk. With this in mind, *mesh partitioning* has been an important research topic for many years [30]. Partitioning unstructured grid is complicated by cells that span partition boundaries. Data associated with such cells may be duplicated among several partitions, but at the expense of extra storage.

10.2 Data Retrieval

Prefetching and caching methods have been investigated for years [51, 74, 18] and has long been used to speed up execution both at the system and application level. Over the years, the computing community have tried to improve system throughput by integrating prefetching into hardware and software systems [23, 16]. The filesystem cache also prefetches pages following an explicitly accessed page in the hope that the prefetched page will be accessed next and reads to disk will be reduced. However, filesystem prefetching will increase the number of inappropriate pages loaded when the user does not proceed through the file in the manner the filesystem predicts, which degrades performance. However, these approaches do not readily lend themselves to spatial data because they view the dataset as one-dimensional, missing important neighborhood relationships available in an n-dimensional view of the dataset. *Iteration Aware Prefetching (IAP)*, first described in [56] is unusual both because it maintains a spatial view of the data, and because it uses advance knowledge of the access

pattern expressed as an iterator.

10.3 Data Declustering

Spatial multidimensional declustering has been the focus of considerable research [17]. A recent work by Rusu et al. [60] provides an extensive overview of array systems and declustering schemes. Some of the early work includes declustering based on hilbert curve [26], disk modulo [45] and cyclic declustering [53]. In the area of multidimensional declustering, Lo et al. [38] propose a scheme that is optimal in share nothing environments for partial match queries. Li et al. [35] created a method that achieves optimum parallelism for high range multidimensional data if the data distribution is uniform in each dimension. This is not practical for unstructured grids and thus, not directly applicable to unstructured grids. Most of the existing work on spatial data is focused on structured multidimensional datasets. The work by Ray et al [54] presents a mechanism for declustering spatial data and speeding up spatial join queries. The approach uses structured data stored in a relational database. Recent work [71] in multidimensional and spatial data declustering takes advantage of the query history to improve the declustering mechanism. Data replication across disks is however inherent in this approach.

CHAPTER 11

CONTRIBUTION

One of our main contributions is the reorganization of unstructured grids into partitions. Access times required before reorganization were impractically large because reading data for a certain region in the spatial data results to a considerable amount of hopping around on disk. We reorganize the data and merge the resulting files to improve the locality of reference within the one dimensional file. Even without the merging step, preliminary testing shows that the improvement in locality provided by partitioning the dataset into cell groups brought traversal times into feasible range, from more than half a day down to an hour and a half.

Different specialized tools have been developed for handling big data. However, available tools don't take advantage of some characteristics of the data such as the size and location, to create unique access and storage pattern for the data. The HDFS file system [15] used in Hadoop [78] for example, is more concerned about data availability and uses replication to achieve this feat. We are more concerned with preventing unnecessary duplication and improving data retrieval performance. We could choose to add replication for enhanced availability on top of our existing mechanism, if desired.

We create a mechanism for efficiently storing and retrieving data and characterize the memory cost and the performance. This helps answer important questions such as; Given a storage cost (overhead) a user is willing to pay, what is the best performance that can be

achieved?

Conceptual inputs and outputs are shown below:

Inputs:

- Storage Overhead
- Access Pattern

Outputs:

- Performance
- Partitioning

Our main contributions are:

1. We dramatically enhance I/O performance with unstructured meshes (a.k.a grids).
2. Improve locality of reference by reorganizing large files using the rod storage model for unstructured grids.
3. Efficiently handle simplices that span chunk boundaries without duplicating data values.
4. Successfully apply a prefetching cache takes advantage of prior knowledge of the access pattern and further improves performance.
5. We investigate the relationship between the partition size, the access pattern and the size of available memory and the effects on the overall system performance.
6. We investigate in memory partitioning of unstructured grids and its effect on performance

7. Successfully apply the Disk Modulo (DM) declustering mechanism to unstructured grids and analyze the retrieval performance when we apply Iteration Aware Prefetching (IAP)

8. Reduce retrieval costs and achieve efficient load balancing when using smaller partitions by paying bandwidth costs once while performing parallel disk reads

9. We provide big data users with an efficient method for partitioning large unstructured grids given a user specified access pattern

10. We create a mechanism that can predetermine the best way to partition a dataset for optimal storage and retrieval performance.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] AmazonRedshift webpage. Amazon Web Services, http://aws.amazon.com/redshift/.

[2] FITS webpage. `http://fits.gsfc.nasa.gov/`.

[3] HDF5 webpage. `http://hdf.ncsa.uiuc.edu/products/hdf5/index.html`.

[4] NetCDF webpage. `http://www.unidata.ucar.edu/software/netcdf/`.

[5] Unstructured tetrahedral grid of F-16 webpage. `http://tetruss.larc.nasa.gov/apps/f16.html`.

[6] Vertica webpage. HP Vertica Analytics, http://www.vertica.com/.

[7] Visualization of earthquake simulation webpage. `http://vidi.cs.ucdavis.edu/research/unstructuredgrid`.

[8] Graphics Gems Repository. http://www.graphicsgems.org/, January 2014.

[9] Oyindamola O. Akande and Philip J. Rhodes. Iteration Aware Prefetching For Unstructured Grids. In *Proc. IEEE International Conference on Big Data*, pages 219–227, 2013.

[10] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, page 8. ACM, 2005.

[11] Ibraheem Al-Furaih and Sanjay Ranka. Memory hierarchy management for iterative graph structures. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 298–302. IEEE, 1998.

[12] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. In *ACM SIGMOD International Conference on Management of Data*, number EPFL-CONF-198683, 2014.

[13] Nihat Altiparmak and Ali Şaman Tosun. Generalized optimal response time retrieval of replicated data from storage arrays. *ACM Transactions on Storage (TOS)*, 9(2):5, 2013.

[14] CB Barber and H Huhdanpaa. Qhull. *The Geometry Center, University of Minnesota, http://www. geom. umn. edu/software/qhull*, 1995.

[15] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project. http://hadoop. apache. org/common/docs/current/hdfs_design. pdf*, 2008.

[16] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):40–52, 1991.

[17] Chung-Min Chen and Christine T Cheng. From discrepancy to declustering: near-optimal multidimensional declustering strategies for range queries. *Journal of the ACM (JACM)*, 51(1):46–73, 2004.

[18] F. Chen, X. Ding, and S. Jiang. Exploiting disk layout and block access history for i/o prefetch. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, page 201, 2010.

[19] Ling Tony Chen, R Drach, M Keating, S Louis, Doron Rotem, and Arie Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. *Information Systems*, 20(2):155–183, 1995.

[20] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*, pages 191–198. IEEE, 2005.

[21] Paolo Cignoni, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. External memory management and simplification of huge meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 9(4):525–537, 2003.

[22] Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, Entong Shen, and Ting Yu. Differentially private spatial decompositions. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 20–31. IEEE, 2012.

[23] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential hardware rrefetching in shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 6(7):733–746, 1995.

[24] HC Du and JS Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Transactions on Database Systems (TODS)*, 7(1):82–101, 1982.

[25] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

[26] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *In Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[27] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[28] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[29] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[30] George Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 56–66. ACM, 2003.

[31] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[32] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, R. Latham, T. Peterka, M. Papka, and R. Ross. Towards parallel access of multi-dimensional, multi-resolution scientific data. In *Proceedings of 2010 Petascale Data Storage Workshop*, 2010.

[33] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[34] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *ACM Sigmod Record*, 30(1):19–24, 2001.

[35] Jianzhong Li, Jaideep Srivastava, and Doron Rotem. Cmd: A multidimensional declustering method for parallel database systems. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 3–14. Citeseer, 1992.

[36] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 8(3):239–254, 2002.

[37] Yu-lung Lo, Kien A Hua, and Honesty C Young. A general multidimensional data allocation method for multicomputer database systems. In *Database and Expert Systems Applications*, pages 357–366. Springer, 1997.

[38] Yu-Lung Lo, Kien A Hua, and Honesty C Young. Gemda: A multidimensional data partitioning technique for multiprocessor database systems. *Distributed and Parallel Databases*, 9(3):211–236, 2001.

[39] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: Reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[40] Rainald Löhner and Paresh Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10):1135–1149, 1988.

[41] Kwan-Liu Ma, Aleksander Stompel, Jacobo Bielak, Omar Ghattas, and Eui Joong Kim. Visualizing very large-scale earthquake simulations. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 48–61. IEEE, 2003.

[42] Roger MacNicol and Blaine French. Sybase iq multiplex-designed for analytics. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1227–1230. VLDB Endowment, 2004.

[43] DJ Mavriplis. Unstructured grid techniques. *Annual Review of Fluid Mechanics*, 29(1):473–514, 1997.

[44] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. *arXiv preprint arXiv:1402.3281*, 2014.

[45] Bongki Moon and Joel H Saltz. Scalability analysis of declustering methods for multidimensional range queries. *Knowledge and Data Engineering, IEEE Transactions on*, 10(2):310–327, 1998.

[46] Dylan Nelson, Mark Vogelsberger, Shy Genel, Debora Sijacki, Dušan Kereš, Volker Springel, and Lars Hernquist. Moving mesh cosmology: tracing cosmological gas accretion. *Monthly Notices of the Royal Astronomical Society*, 429(4):3353–3370, 2013.

[47] E.J. Otoo, D. Rotem, and S. Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 25–32. ACM, 2007.

[48] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 214–221. ACM, 1993.

[49] S. Papadomanolakis, A. Ailamaki, J.C. Lopez, T. Tu, D.R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM, 2006.

[50] Jignesh M Patel and David J DeWitt. Partition based spatial-merge join. In *ACM SIGMOD Record*, volume 25, pages 259–270. ACM, 1996.

[51] R. Hugo Patterson, Garth A.Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM Symp. on Operating System Principles*, pages 79–95, Copper Mountain Resort, CO, Dec. 1995.

[52] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[53] Sunil Prabhakar, Khaled Abdel-Ghaffar, Divyakant Agrawal, and Amr El Abbadi. Cyclic allocation of two-dimensional data. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 94–101. IEEE, 1998.

[54] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. A parallel spatial data analysis infrastructure for the cloud. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 274–283. ACM, 2013.

[55] Philip J. Rhodes and Sridhar Ramakrishnan. Iteration aware prefetching for remote data access. In H. Stockinger, Rajkumar Buyya, and Ron Perrott, editors, *Proc. 1st International Conference on e-Science and Grid Computing*, pages 279–286, 2005.

[56] Philip J. Rhodes, Xuan Tang, R. Daniel Bergeron, and Ted M. Sparr. Iteration aware prefetching for large multidimensional scientific datasets. In *SSDBM'2005: Proc. of the 17th international conference on Scientific and statistical database management*, pages 45–54, Berkeley, CA, US, 2005. Lawrence Berkeley Laboratory.

[57] Philip J. Rhodes, Xuan Tang, R. Daniel Bergeron, and Ted M. Sparr. Out of core visualization using iterator aware multidimensional prefetching. In *Proc. SPIE*, volume 5669, pages 295–306. Visualization and Data Analysis, 2005.

[58] Kenneth A Ross and Evangelia Sitaridi. Partitioned blockmap indexes for multidimensional data access. Technical report, Department of Computer Science, Columbia University, 2012.

[59] D. Rotem, E.J. Otoo, and S. Seshadri. Chunking of large multidimensional arrays. *Lawrence Berkeley National Laboratory*, 2007.

[60] Florin Rusu and Yu Cheng. A survey on array storage, query languages, and systems. *arXiv preprint arXiv:1302.0103*, 2013.

[61] Hans Sagan. *Space-filling curves*, volume 18. Springer-Verlag New York, 1994.

[62] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms–ESA 2011*, pages 469–480. Springer, 2011.

[63] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 328 – 336, Houston, TX, USA, 1994. IEEE Computer Society.

[64] Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[65] E. Shriver, C. Small, and K.A. Smith. Why does file system prefetching work. In *Proceedings of the 1999 USENIX Annual Technical Conference*, volume 27, 1999.

[66] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. Edo: Improving read performance for scientific applications through elastic data organization. In *Proceedings of the IEEE International Conference on Cluster Computing, ser. Cluster*, volume 11, 2011.

[67] Y. Tian, S. Klasky, W. Yu, H. Abbasi, B. Wang, N. Podhorszki, R. Grout, and M. Wolf. Smart-io: System-aware two-level data organization for efficient scientific analytics. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 181–188. IEEE, 2012.

[68] Y. Tian and P.J. Rhodes. Partial replica selection for spatial datasets. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–10. IEEE, 2012.

[69] Yun Tian and Philip J. Rhodes. A fast location service for partial spatial replicas. In Shantenu Jha, Nils Felde, Rajkumar Buyya, and Gilles Fedak, editors, *Proc. 12th IEEE/ACM International Conference on Grid Computing (GRID 2011)*. IEEE Computer Society, September 2011.

[70] Frank SC Tseng, Yen-Hung Kuo, and Yueh-Min Huang. Toward boosting distributed association rule mining by data de-clustering. *Information Sciences*, 180(22):4263–4289, 2010.

[71] Ata Turk, Kerim Yasin Oktay, and Cevdet Aykanat. Query-log aware replicated declustering. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):987–995, 2013.

[72] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 3(4):370–380, 1997.

[73] Gino Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[74] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of USENIX Annual Technical Conference*, 2009.

[75] Wei Wang, Jiong Yang, and Richard Muntz. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, volume 97, pages 186–195, 1997.

[76] Yijian Wang and David Kaeli. Profile-guided file partitioning on beowulf clusters. *Journal of Cluster Computing, Special Issue on Parallel I/O*, 2006.

[77] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, page 12. ACM, 2013.

[78] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[79] K. Wu, S. Ahern, EW Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, et al. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, volume 180, page 012053. Institute of Physics Publishing, 2009.

[80] Min Zhou, Onkar Sahni, Ting Xie, Mark S Shephard, and Kenneth E Jansen. Unstructured mesh partition improvement for implicit finite element at extreme scale. *The Journal of Supercomputing*, 59(3):1218–1228, 2012.

VITA

Oyindamola Akande received a Bachelors of Science from Bowen University in Iwo, Nigeria in 2007 graduating with first class honors. He worked briefly in the financial industry in Nigeria before proceeding for post graduate studies. He earned his Masters in Computer Science from the University of Mississippi (UM) in 2010 and began his Ph.D. studies at UM after completing his masters degree.

At the University of Mississippi, Oyindamola worked as a Research Assistant at the Institute for Advanced Education in Geospatial Sciences (IAEGS), and with Dr Rhodes working on large spatial unstructured data. During his Ph.D. program, Oyindamola interned several times in Silicon Valley working as a Software Engineer with Intel Corporation.

His research interests include spatial data, scientific computing, cluster computing and distributed computing systems.